AD-A035 915 COMPUTER SCIENCES CORP HUNTSVILLE ALA F/G 9/2
LANGUAGE CONTROL FACILITY (LCF) STUDY. EVALUATION OF THE SOFTWA--ETC(U)
DEC 76 W ANDERSON, T L DUNBAR, P L BELFORD F30602-76-C-0024
UNCLASSIFIED CSC-4549-110 RADC-TR-76-386-VOL-2 NL

1 OF 2
AD-A
035 915

AD-A035 915

LANGUAGE CONTROL FACILITY (LCF) STUDY

EVALUATION OF THE SOFTWARE TOOLS FOR THE LCF

COMPUTER SCIENCES CORPORATION
HUNTSVILLE, ALABAMA

DECEMBER 1976

RADC-TR-76-386, Vol II (of two)
Final Technical Report
December 1976

LANGUAGE CONTROL FACILITY (LCF) STUDY
Evaluation of the Software Tools For The LCF

Computer Sciences Corporation

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-76-386, Volume II (of two)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) LANGUAGE CONTROL FACILITY (LCF) STUDY Evaluation of the Software Tools for the LCF		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 Oct 75 - 1 Jul 76
		6. PERFORMING ORG. REPORT NUMBER 4549-110
7. AUTHOR(s) William Anderson Peter L. Belford Terry L. Dunbar Richard Gilinski		8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Corporation 515 Sparkman Drive Huntsville AL 35806		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 55500821
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE December 1976
		13. NUMBER OF PAGES 181
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)
Same

18. SUPPLEMENTARY NOTES
RADC Project Engineer: Samuel A. DiNitto Jr. (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Higher Order Languages, Language Control Facility, JOVIAL, JOCIT, Compiler, Compiler Validation, Program Verification, Statistics Collection, SEMANOL, Language Specification, Network, JAVS, AUTOVON, ARPANET, Software, Staffing.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
This effort was undertaken to identify the necessary facilities, operating procedures, personnel, and software tools to effectively control the proliferation of Higher Order Languages for computer programming. This control would allow the Air Force (and all of DOD) to standardize on the use of a few well controlled programming languages. This in turn would promote the use of Higher Order Languages over Machine Oriented Languages with the added advantage of smaller development and maintenance costs for software, reduced

training costs for programmers, and increased transferability of both software and programmers.

This report has been reviewed and is approved for publication.

APPROVED: [illegible]

SAMUEL A. DINITTO, Jr.
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1 - INTRODUCTION

1.1 OBJECTIVES OF STUDYING THE SOFTWARE TOOLS

This volume of the report deals with the software tools required to support a Language Control Facility (LCF). The report includes an analysis of these tools, and recommendations for the use and/or enhancement of them for the LCF application. Five software tools are analyzed: (1) a compiler implementation tool (JOCIT), (2) a language utilization measurement tool (JLMT), (3) a compiler validation system (JCVS), (4) an automated program verification system (JAVS), and (5) a language specification tool (SEMANOL).

1.2 DOCUMENT ORGANIZATION

Section 1 provides an introduction to this volume. The remainder of the volume is concerned with an analysis of the support software tools and a discussion of the recommended tools for the LCF.

In Sections 2 through 6, a brief description of each tool is followed by an analysis of how well it has met its original design requirements and how relevant it is to the LCF concept. Section 2 discusses the JOVIAL Compiler Implementation Tool, JOCIT, which is aimed at providing low cost and efficient JOVIAL compilers for the J-3 language. Section 3 discusses the design of a statistics-gathering methodology known as the JOVIAL Language Measurement Tool (JLMT). Section 4 addresses the existing JOVIAL Compiler Validation System (JCVS) which has been implemented for both the J-3 and J73 Level I languages. Section 5 discusses the JOVIAL Automated Verification System (JAVS) implemented for JOVIAL/J-3 users as a companion to the JOCIT Compiler under the HIS-6000 series machines with the GCOS operating system. Section 6 presents the results of studying the SEMANOL system, a programming tool for describing both the syntax and the semantics of an HOL, as a machine-verifiable language specification.

Sections 7 through 14 present recommendations for using the tools within the LCF.

Recommendations for tools are introduced in Section 7, and the functioning of the tools within the LCF is discussed in Section 8. Section 9 presents recommendations for enhancing the JOCIT program to meet the LCF rehosting and retargeting requirements, the integration of other tool functions into the compiler, and the extension of the compiler building tool concept to other HOLs. Section 10 discusses the problem of language specification. Section 11 presents functional design requirements for a statistics gathering function within the LCF. Section 12 presents recommendations for enhancing the JCVS tests to meet the LCF compiler validation requirements. Section 13 discusses the incorporation of JAVS into the LCF and how the compiler, statistics collector, and program validator may be integrated into a coherent package, employing common interfaces. Section 14 closes this part of the report by presenting recommendations for support tools to assist in program development, debugging, and HOL translation.

These sections also discuss how the tools function together within the LCF, and where tools are considered deficiencies. Recommendations are made for incorporating a compiler implementation tool, statistics-collector, compiler validator, and a program validator. In addition, these sections discuss how these HOL software tools are to be supplemented by an HOL language specification and certain support tools to assist in other aspects of program and system development using the HOL. The sum of Sections 7 through 14 is a set of software functional design requirements for an LCF. Since the majority of the tools already implemented support the JOVIAL/J-3 language, most of the discussion is J-3 oriented. However, the presentation is supplemented with design recommendations for broadening the tools' applicability to other HOLs.

SECTION 2 - JOCIT

2.1 BACKGROUND

The original intent of the JOCIT effort was to develop a compiler building tool for the J73 language. When the contract was awarded, the J73 language still was not firmly defined, and after a few weeks of study of the language it was determined that the development costs were higher than originally estimated. Therefore, in order to produce a more useful product that demonstrated the principal elements of the original objectives, the project was redirected towards developing a J-3 compiler building tool. Since the need within the Government for a J-3 capability was sufficiently real and urgent, the new project goals were highly practical. Inc ed, the JOCIT J-3 compiler developed on the WWMCCS machine (HIS-6000) has been adopted as a standard by two significant Air Force projects.

The principal objectives of the JOCIT J-3 development were to:

- Reduce the time and cost of implementing and maintaining JOVIAL J-3 compilers
- Ensure that JOVIAL language sets implemented on different computers are consistent
- Enable the rapid inclusion of any new JOVIAL features into every compiler built with the tool, including those compilers implemented before the feature was accepted
- Enable the compilers built with the tool to incorporate modern optimization techniques that overcome many forms of poor programming

Although the redirection to produce a running, debugged, efficient, and reliable J-3 compiler necessarily had the effect of diluting some of the goals of the tool, nevertheless, the objectives were largely met. The most objectively measurable result of the JOCIT effort was the development of a production-grade J-3 compiler currently in heavy operational use. However, it is somewhat difficult to assess the result of the tool development in equally objective terms, because no new retargeting or rehosting effort has been undertaken. The following paragraphs describe the essential features of the JOCIT program and assess the relative success in meeting the stated goals.

2.2 DESIGN FEATURES

The following design elements were incorporated into JOCIT:

- Target-machine-dependent code isolated into functional modules
- Global optimization techniques to meet the requirements of language independence, host-machine independence, and target-machine independence
- The GENESIS system used for writing the J-3 language specification, the resulting tabular form of which is processed by a language-independent analyzer program
- A prototype compiler to compile the full J-3 language; the prototype can be used as a model for rehosting the J-3 version or as a basis for building a J73 JOCIT
- Over 95 percent of the JOCIT code written in an HOL (SYMPL); use of machine code is restricted to host-machine-dependent interfaces

2.3 CHARACTERISTICS OF THE JOCIT J-3 COMPILER

JOCIT embodies the following three features which, together, realize the goal of a tool for the generation of standard JOVIAL J-3 compilers:

- JOCIT is a stable, well-debugged, efficient, production-quality J-3 compiler. It realizes the most advanced optimization in any

JOVIAL J-3 compiler to date, and even though the compiler is large (40-50K words of HIS-6000 main memory), it is quite fast and generates extremely informative and useful listings.

- Retargeting of JOCIT is a known, relatively straightforward, but not trivial process. It is achieved through total replacement or partial modification of certain compiler modules, and the installation of the JOVIAL library on the new target machine.

- Rehostability of JOCIT is achieved principally by programming the JOCIT modules in SYMPL. Rehosting is considerably more complex than retargeting, but the steps are well understood and meet the tool requirement by costing only a fraction of a comparable, totally new compiler implementation.

These three considerations are addressed in the following subsections.

2.3.1 USER INTERFACE

The JOCIT J-3 compiler is operated in a standard fashion entirely compatible with other GCOS language processors. That is, the command syntax and file specifications conform to GCOS standards, and the JOCIT user is required to learn only the computer options in order to invoke the compiler.

2.3.2 THE J-3 LANGUAGE

JOCIT implements the full J-3 language, with certain extensions added to satisfy unique customer requirements (including a special source language I/O facility to satisfy a user requirement for compatibility with the nonstandard Honeywell J-3 compiler). The diagnostic capability is thorough, and extensive use is made of parameterized diagnostics (for example, providing for insertion of identifier or reserved word names).

2.3.3 COMPILER LISTINGS

The JOCIT compiler provides a comprehensive set of compiler listings. These listings include interspersed Phase I diagnostics; a consistent diagnostic format for all phases; an extensive object program assembly language-format listing identical to GMAP, a complete "set-used" listing for all program constructs (define names, status constants, program variables, labels, procedures, etc.); and a program environment listing.

2.3.4 OBJECT PROGRAM EFFICIENCY

The JOCIT program expends much effort to obtain object program efficiency. This has been achieved through two means: global, target-machine-independent optimization, and a code generation scheme that optimizes register usage and performs considerable special case analyses. Global optimization includes:

- Elimination of redundant common expressions computations
- Redistribution of loop-constant code
- Reduction of formal loop operator strength
- Improvement of compile-time constant arithmetic and subexpressions (e.g., elimination of multiplications by 1)
- Recognition of dead code
- Evaluation of compile-time constant predicates (e.g., AA = 1$...IF AA$)

All computational memory is embodied in the optimizer-code generator file (IL) interface, while local optimizations are performed by the code generator to produce the optimum sequence for each recognizable case. The combination of global and local optimization, which attempts to minimize generated code space, is successfully realized in the JOCIT J-3 compiler. Further improvements that would have a high payoff in production programs are:

- Regional index register dedication
- Loop control variable (LCV) index register dedication
- Improved strength reduction, including test replacement and dead LCV elimination

- Dead variable analysis
- Code straightening

Of the 60 percent improvement in generated code over the previous HIS J-3 compiler, 10 to 15 percent is attributable to global optimization, while the balance is derived from the local code generation algorithms. Even though there is room for improvement, the level and quality of the realized optimization are the notable achievements of the JOCIT J-3 model.

2.3.5 COMPILER EFFICIENCY

Considering its optimizing capabilities, the JOCIT J-3 compiler is comparatively fast. However, the compiler's instantaneous main memory requirements are quite high; 40K words is the minimum partition plus the size required for the compiled program's symbol table. The compiler already is heavily segmented into separate overlay loads. Only a radical redesign could reduce the core requirements, and only at the cost of severely reduced compiler speeds. The large size of the compiler is due to the following reasons (listed in descending order of impact):

- Complexity - The JOCIT J-3 compiler was designed for maximum user utility. It performs an enormous number of complicated tasks in order to produce pinpoint diagnostics, to perform global flow analysis/optimization, to pack tables optimally, and to provide a sophisticated and useful COMPOOL facility. The augmented J-3 language it compiles is huge, and the code generator achieves its goals through complex algorithms that require a considerable number of source code lines to effect. It is doubtful that the number of lines of code in the compiler itself could be materially reduced without seriously compromising user convenience and object code performance.

- Compiler Architecture - The compiler uses a minimum number of phases and intermediate files and requires a large symbol table to be resident throughout the compilation process. It is conceivable that by partitioning the compiler into more functional phases (a multipass code generator is a possibility) the maximum phase size could be reduced; but, as pointed out earlier, compiler speed would be reduced and additional program complexity (more intermediate files, for example) would result.
- Use of HOL - Because the JOCIT J-3 compiler is written in SYMPL and compiled by a small compiler which incorporates only a modest number of local optimizing algorithms, the JOCIT compiler contains more lines of object code than would be the case if it had been written in assembly code or compiled by a sophisticated, optimizing SYMPL compiler. A more compact compiler also could be achieved by rewriting the JOCIT compiler using J-3, thereby obtaining the benefits of the compiler's own optimization. However, the J-3 language is much less suited to compiler implementation than is SYMPL and carries excess baggage (e.g., fixed point arithmetic, lengthy prologues and epilogues) that is costly and unnecessary. Optimizations could be added to the SYMPL compiler to reduce object code without unduly compromising rehostability or retargetability of either SYMPL or JOVIAL.

2.3.6 DEBUGGING (USER)

The inclusion of the MONITOR statement and ENCODE/DECODE provide considerable debugging convenience for the user. In particular, the availability of the compiler command option to suppress compilation of all MONITOR statements allows the user the convenience of retaining his MONITOR statements in the source program without paying the compilation - and resulting object program - price.

2.3.7 DEBUGGING (COMPILER MAINTENANCE)

The JOCIT model includes a wide range of built-in compiler debugging features, mostly in the form of formatted table and file dumps that can be selected individually during maintenance-mode execution of the compiler. The debugging routines themselves occupy symbol table space and are overwritten by symbol table entries during production-mode compilation. Thus, the production mode compiler is not enlarged since the maintenance debugging routines are not ordinarily present; however, full debugging capability is available in the compiler by exercising an option.

2.3.8 RELIABILITY

The reported error rate on the production JOCIT J-3 compiler is comparatively low. The errors tend to be distributed throughout the compiler modules, and it is rare for the compiler to fail completely. Not surprisingly, the most vulnerable phase of the compiler is the optimizer since very large programs with complex flow can cause the optimizer to abort. However, in keeping with the diagnostic approach of the design, these failures are (almost without exception) self-detected anomalies. Most optimizer failures relate to unnecessarily complex space management functions which are subject to unpredictable, subtly-compounded errors. The forthcoming JOCIT improvements project, which provides for considerable optimization enhancement, will include simplified restructuring of the optimizer data base and space manager to strengthen this area considerably.

2.4 <u>RETARGETING</u>

In order to achieve retargeting of the present JOCIT model, the following steps are required:

1. Install library on target machine.
2. Adjust compiler code for target machine sensitivity.
3. Write new direct code processor.

4. Write new code generator.

5. Modify the editor phase to produce object code listing in new target-machine format.

6. Add new object module formatter.

7. Provide for translation of host-machine constant formats to target-machine formats.

These steps are discussed in detail in the following paragraphs.

2.4.1 LIBRARY INSTALLATION

The J-3 library consists mainly of I/O and ENCODE/DECODE routines, string routines, and MONITOR routines. Retargeting requires rewriting these routines for each new target. Except for the MONITOR routines written in SYMPL, these routines are written in assembly code that is not directly transferable to a new target machine. Installation of the library is not a simple task since even the MONITOR routines must be rewritten (unless, of course, a SYMPL compiler exists for the new target machine).

2.4.2 ADJUSTMENT FOR TARGET-MACHINE SENSITIVITY

Many routines within the compiler are affected by various characteristics of the target machine. These are partly parameterized through use of a target-machine descriptor block (COMMON Block COM08T). However, this parameterization is not yet complete. Typical parameters of interest are:

- Target word size
- Target byte size (bits per byte)
- Target bytes per word
- Maximum and minimum integer values
- Maximum and minimum floating values
- Medium packing access field descriptions
- Addressing units per word

- Character set internal representation
- Target numeric-value representations

The following routines within the compiler presently must be adjusted as described below:

ALOCTR	Object Program Data Allocator - Describe medium packing and type characteristics.
XREF	Cross-Reference Lister - Tailor target-dependent listing.
CCP	Compiler Control Program, i.e., control card scanner - Recognize multiple target option.
JXEC	Compiler Executive or "cradle" - Sequence the compiler as necessary for different target-dependent phases (e.g., the code generator).
COM08T	Target Parameter Data Block - Modify target-machine parameters.
PCON	Constant Posting Routine - Modify as necessary to reflect different internal forms for target.
JINIT	Initialization of Compiler - Post target-specific intrinsic functions, if any (for example, the correct library routine entry point name for the string routines, I/O routines, etc.).
JPF1	Pass 1 Analysis Pragmatic Functions - Convert source form constants to target form.
PF1PR1	Preset Processing Subroutine of Pass 1 Pragmatic Functions - Prepare preset constants in target format.
OPT2A	Pass 2 Optimizer Constant Arithmetic Routine - Modify constant arithmetic to manipulate target form values.

Most of these modifications are individually trivial; however, the number of different routines to be examined and modified makes the composite task moderately complex.

2.4.3 NEW DIRECT CODE PROCESSOR

The direct code processor must be rewritten for each new target-machine assembly language format. This processor is a functionally separate module which must be replaced in the link-edit of the first analysis phase. This necessitates target-machine sensitivity in JXEC to load the proper phase, and requires the maintenance of a unique analysis Phase 1 for each target supported (all but the direct code processor within the phase have the same code for each target machine).

2.4.4 NEW CODE GENERATOR

The major modification for retargeting is the writing of a new code generator. If the level of local optimization and the effective realization of the global optimizations performed by the optimizer are to be sustained, a substantial effort is required.

The basic architecture of the current HIS-6000 code generator may be retained (which considerably reduces the design effort), and much of the machine-independent code (e.g., the triad table builder) need not be rewritten. Still, this must be considered a major task. There will be one code generator for each target machine; JXEC will select the appropriate code generator phase.

2.4.5 EDITOR PHASE MODIFICATION

The editor phase must be modified (there will be a unique editor for each supported target) to generate the proper assembly-like object code listing. The preset-constant processing also must be modified to align values in a manner consistent with the target machine characteristics.

2.4.6 OBJECT MODULE FORMATTER

An object module formatter (actually a part of the editor phase) must be written for each target. The scope of this task is a function of both the complexity of the object module format requirements and the reliability and clarity of the system documentation that describes it. This can range from relatively straightforward to extremely arduous.

2.4.7 TRANSLATION TO TARGET CONSTANT FORMAT

This process has been identified in preceding paragraphs. The problem is to convert from the compiler's internal constant representation (HIS-6000 format) to the target machine format. This requirement affects several compiler modules. For example, when the optimizer performs compile time comparisons between character constants, correct inequalities may be computed only when using the target representation, since its collating sequence may not be the same as the HIS (host) character representation.

2.5 REHOSTING

Moving a compiler from one machine to another is a complex and demanding task. Within the constraints of the project noted earlier, the JOCIT program was implemented with rehosting as an important but not overriding design criterion. Thus, when two criteria stood in conflict - often the case during the short term of the project - the production compiler objectives governed.

Rehosting of JOCIT is not an easy task, but the rehosting procedure is well understood and has been tried; the JOCIT model was developed on the 1108 and subsequently rehosted to GCOS. This effort was much more than just a rehosting, however, so it cannot stand as a wholly objective measure of the rehostability of the design. Three major elements of the compiler did prove to be portable; these were the GENESIS-based analysis phase, the optimizer, and the SYMPL compiler. The first two were extensively modified, so that subsequent rehostability is not directly inferrable from that experience. However, as modifications were applied, rehosting considerations influenced the design, so that these programs are expected to be even more portable than their predecessors.

As used in this section, rehosting means moving the JOCIT J-3 compiler model to a new compiling-machine environment. Rehosting in this sense does not entail retargeting for the same machine; this is considered separately. The minimum rehosting steps are:

1. Installing the JOCIT executive on the new host.
2. Retargeting the SYMPL compiler for the new host (the SYMPL compiler itself need not be rehosted).
3. Adapting the compiler data base to the new host.
4. Reformatting diagnostic messages for the new host.
5. Modifying GENESIS to generate tables for the new host.
6. Tailoring constant (including preset constant) processing to the new host formats.
7. Recompiling all JOCIT modules including GENESIS output (which is in the form of SYMPL source statements) through the retargeted SYMPL compiler.
8. Linking and integrating the recompiled modules on the new host.
9. Testing the integrated compiler using a selected target (perhaps the new host itself) as the testing medium.

These rehosting activities are amplified in the following paragraphs.

2.5.1 INSTALLING THE EXECUTIVE

All of the machine-dependent and operating system-dependent code is isolated in a set of modules known collectively as the compiler executive (or cradle, as it is sometimes called). To achieve maximum compiler performance, and to fit the compiler's executive requirements most efficiently to the host machine, these routines were written entirely in host-machine assembly language (GMAP). The executive requirements are extensive and support both the JOVIAL and SYMPL compilers, which are integrated into a single link-edited program. The principal executive functions are:

- Control card processing
- Compiler phase selecting, sequencing, and loading
- Buffer management and file control block definition
- Space management
- All file management and I/O operations
- Listing control
- Intermediate file debugging dumps and control
- Exception processing

Although the code in the executive is a small percentage of the code in the compiler, it must be considered "difficult" code, since it deals intimately with the host operating system conventions. Rehosting the executive begins with a detailed analysis of the host system services, itself a substantial effort. Actually, this effort can be simplified by existing familiarity with the new host system or by simplified or well-documented interfaces. On the other hand, to retain the present level of executive efficiency, this effort may be expected to be moderately demanding at best. Where efficiency is not a consideration - this must be specified in the rehosting requirements - the effort may be simplified by writing in a language such as FORTRAN. Still, the executive embodies many interfaces, both on the compiler side and on the system side, and this portion of the rehosting task is a substantial effort.

2.5.2 RETARGETING SYMPL

A code generator must be added to the SYMPL compiler to generate code for the new host machine. This is the principal bootstrapping mechanism. The generator must be supplemented by an object-module formatter and a code-listing editor. At a minimum, these are moderate efforts. This has been proved by modification of the SYMPL compiler numerous times in the past. The level of effort is largely determined by the sophistication of the local optimizations attempted in the generator, and this is dictated by the efficiency requirements of the rehosted compiler.

2.5.3 ADAPTING THE COMPILER DATA BASE

The compiler data base consists of numerous data declarations written in SYMPL. Since at the time JOCIT was developed the SYMPL compiler required programmer specification of array item packing, adapting the data base requires rewriting these declarations. The difficulty of this problem may be lessened in two ways: (1) rehosting to a machine whose word size is not less than the HIS-6000 word size (36 bits) and (2) parameterizing the data base declarations. For example, if JOCIT were to be rehosted on the UNIVAC 1100-series machines whose word size is 36 bits, the adaptation effort is approximately zero. (It may be greater than zero if an attempt is made to optimize part-word references into half, third, and sixth-words, which are directly accessible on the 1100-series.) Some parameterization of the data declarations already exists; the optimizer phase, for example, is parameterized for any host machine word size of 16 to 60 bits. Adaptation also may require restructuring of the compiler common data for a host that does not support the named COMMON facility.

2.5.4 DIAGNOSTIC MESSAGES

The diagnostic messages in the compiler are coded using a special GMAP macro in order to make insertion of new diagnostics extremely simple. For a new host, a similar macro (or equivalent) must be developed, and the messages rewritten. This is an isolated, relatively simple task.

2.5.5 GENESIS MODIFICATIONS

The output of GENESIS is in the form of SYMPL source language array and status declarations, switches, and preset data. The only required change to GENESIS is to allow for generation of table declarations to conform to the host machine word size. GENESIS already accommodates 32-bit, 36-bit, and 60-bit hosts; for word sizes not included in this set, modifications must be made.

2.5.6 CONSTANT PROCESSING

Code within the JOCIT model must be examined for sensitivity to HIS-6000 constant representation and collating sequence. Where there is implicit assumption about host format, this must be parameterized. It is anticipated that such sensitivity affects very few compiler modules. The primary concern in internal constant representation is for retargeting, especially where target word size exceeds host word size. For example, this problem may be solved best by a canonical constant representation from which any target representation may be derived, and by routines which convert back and forth as necessary for constant arithmetic.

2.5.7 RECOMPILING JOCIT MODULES

Once the above tasks are accomplished, the first step in the physical rehosting task is to recompile all SYMPL coded modules through the retargeted SYMPL compiler. (This includes, of course, the SYMPL source output from the GENESIS program.) This activity results in one relocatable object module for each source module.

2.5.8 INTEGRATION

Once the recompilation is complete, the rehosted object modules must be linked on the new host with the compiler executive modules and formed into an executable program. This integration is as complex as the compiler overlay structure and the formats of the relocatable object module required by the link editor. This can range from relatively straightforward on the IBM 370 to quite complex on the UNIVAC 1108. The integration activity also includes some minimal testing to verify the correct working of the major compiler interfaces (source input, object output, intermediate file I/O, debugging, etc.).

2.5.9 TESTING

The final step is the verification of the rehosting. One (or more) of the possible target machines is selected as the verification machine. For greatest convenience, this should be the host itself, although this may not be possible because rehosting does not imply retargeting to the same machine. (SYMPL must be retargeted, but the J-3 compiler itself need not be.) Typically, this verification is achieved using a set of standard tests such as the JCVS series.

2.6 SUMMARY

JOCIT is best described as, first, a competent and serviceable J-3 compiler for the HIS-6000 GCOS machines and, second, a J-3 compiler-building tool. The advantages of JOCIT are:

- Compiler efficiency
- Object code efficiency
- Good diagnostics
- Excellent debugging facilities (both for the user and maintenance team)
- Moderately convenient retargeting
- Use of quick bootstrapping SYMPL compiler tailored to JOCIT needs

The disadvantages are the:

- Size of the compiler
- Changes necessary to make retargeting even less costly
- Changes necessary to make rehosting less costly (a moderately-complex task)
- Reliance on a separate SYMPL compiler that does not take advantage of the JOCIT compiler's own optimization power and requires separate (although rare) maintenance.

SECTION 3 - STATISTICS GATHERING PACKAGE FOR THE JOVIAL LANGUAGE (JLMT)

3.1 BACKGROUND

An approach to gathering statistics for the JOVIAL J73 language is discussed in the Final Technical Report RADC-TR-73-381. This report provides guidelines for measuring JOVIAL principally in terms of usage of features. From these data, ways may be inferred to improve both the language and its compilers. The report asserts that little has been done historically in objective analysis of languages and their use, and it suggests that a rational approach is badly needed to solve the problem of making languages and compilers truly responsive to measured needs.

The study of statistics-gathering for J73 was done against a somewhat moving target since the J73 definition had not completely congealed. Furthermore, there was no J73 compiler available to aid the report writers in supplementing their assertions and methodology with objective data gathering.

The report stresses three principal elements of what the authors call the measurement process:

- Understanding the requirements for measurement and establishing meaningful goals for measurement
- Defining precisely what data are to be gathered, what statistics are to be derived, and what methodology is to be used for acquiring the data
- Statistical analysis of the data, conclusions, and resulting action

The following benefits accrue from measurement:

- There accumulates an objective basis for language revision; i.e., unused features may be deleted, heavily used features may demand optimization concentration, and error-prone features may be simplified or modified

- Within a given application environment, the analysis of language features (density of usage, proneness to error, etc.) helps to determine how effectively the language satisfies the application requirements
- Gathered data may stand to contradict intuitive biases
- User performance profiles may indicate where optimization has its highest payoff
- User diagnostic summaries may show poorly described or inherently difficult language forms which may result in improved documentation
- Diagnostic and program failure statistics may be quite useful for programming management to determine how better to allocate resources when, for example, the data reveal that certain programmers or programming problems show a too-persistent error rate

Throughout the report, the authors use JLMT (JOVIAL Language Measurement Tool) to refer to the model of the statistics-gathering package they describe. JLMT has not been implemented; however, RADC is building a J-3 JOVIAL static statistics gatherer to be run under GCOS on J-3 source modules. It is a stand-alone system that duplicates much of the JOCIT J-3 compiler syntax analysis function, but does not use any of the JOCIT program.

3.2 ANALYSIS OF JLMT

In analyzing the JLMT approach, Final Technical Report RADC-TR-73-381 describes and assesses the following aspects:

- Determination of which data are to be collected
- Data analysis
- Data collection techniques and implementation approaches

The report provides more of a philosophical position; it establishes rough guidelines rather than a blueprint for a JLMT implementation. It is at least one step removed from providing a set of functional specifications for a realizable statistical gathering system.

3.2.1 DETERMINING DATA TO BE COLLECTED

The bulk of the report (Chapter 3) discusses the kinds of data it is most desirable to collect. Included are remarks about how the data might be analyzed for maximum utility. The report identifies three useful classes of data collection:

- Static language usage profiles
- Dynamic user performance profiles
- Compiler performance profiles

These are discussed in the following paragraphs.

3.2.1.1 Static Language Usage Profiles

Static profiles are determined by analyzing source programs and collecting summaries of language usage. The report deals with three kinds of data:

- Run/control information (programmer ID, program ID, etc.)
- Statement statistics (categorizing by statement type, count of symbols, count of blanks, etc.)
- Compilation statistics (size of object program generated, summaries of keywords used, errors, etc.)

The data may be collected meaningfully only by a suitably instrumented compiler. Only a compiler, for example, can determine the size of the object program generated. However, it is quite conceivable to gather statement statistics with a standalone analyzer.

As a guideline for designing a JLMT, the report identifies a number of particular kinds of data that might be collected. Some of these are clearly desirable: for example, statement type, BEGIN-END nesting at statement, categorizing table declarations, and count of referenced external symbols. Others, however, seem of marginal utility or are categorized improperly: LOC, ALL, NWDSEN, SHIFT, and DSIZE, for example, are unlikely to be library routines as they suggest; what the report calls "User Functions" is not defined; DEFINE is improperly classified as a directive; the "size" of a DEFINE formal parameter is not of any interest in a DEFINE statement, since it is by definition limited to a single character; "number and complexity factors in IF statements" is quite vague; and the collection of "pair-wise combination of statements" statistics would greatly inflate the volume of data with a doubtful return in utility.

A large part of Chapter 3 of the report is devoted to bar charts of comparative statistics of different kinds of language form usage. These demonstrate how statistics might be displayed most usefully for analysis. (Since these charts seem to be derived from an actual data gathering activity, but the authors do not reveal the source, it is assumed that they are largely intuitive and given for purposes of exemplifying display techniques.)

In presenting their ideas on the gathering of compilation summary statistics, the authors present a large list of suggested categories. While largely complete, certain items have been omitted: for example, number of instructions generated by instruction mnemonic, number of compiler-generated temps, and count of optimizations performed by type (redundant computations eliminated, branches deleted, code moved, operators reduced in strength, etc.).

3.2.1.2 Dynamic User Performance Profiles

By far, the most demanding aspect of language usage measurement is dynamic or execution statistics collection. There are two distinct problems: (1) gathering the data, and (2) storing the data for subsequent retrieval or analysis. The report presents a short list of execution statistics to be gathered. Both timing and count/frequency data are considered. Timing profiles, where the time is shown as a percentage of total execution, are extremely important for identifying which language constructs, library routines, and user routines are the best candidates for optimization (optimization is used here in the general sense; it does not refer exclusively to compiler-performed optimization , but includes such items as restructuring user algorithms and tuning of library routines).

Relative frequency of dynamic usage of routines seems to be less important than timing data. A routine may be entered quite often but may use little execution time; the question is , how does one respond to this knowledge? Dynamic frequency of language feature usage is, however, important when compared against error propensity for the feature.

The report also addresses a miscellaneous category of statistics including termination conditions, table sizes reached, etc. The inclusion of data on compiler table sizes in this section is clearly misplaced since it is irrelevant to user performance profiles. A discussion of error reporting during dynamic analysis is omitted from the report. Identifying error propensity by language form is an important correlation. Of course, the objective of identifying such a correlation is to refine the language and its documentation and/or language courses. A straightforward correlation - for example, BYTE functions produce a high percentage of execution failure - may be impossible to achieve.

The report also mentions averages such as table size. It is difficult to see how such information could ever be useful; distribution of table sizes would seem to be more important. The fact that the average table size is 150 is less important than the fact that less than 2 percent of all tables exceed a size of 200. In this case, however, it is not clear how the data in either case can be useful.

The report is not clear on how table sizes are to be measured. Clearly, a 100-entry table compiled for the CDC 6600 may occupy fewer words than on the IBM 370, and the analyst must know what the function of this statistic is. Is the analyst to be concerned with the dimensions in the abstract or with its actual storage requirements in the running object program?

Finally, the size of a table is not necessarily described as measured by the maximum index reached during execution. In order to make this data intelligible, it should be supplemented by information describing density of table occupancy. For example, an array might reach an upper bound of 10,000, but be only sparsely occupied; it might be misleading to declare that the size of such an array is 10,000. Instead, where storage size is a problem, the fact that the array is sparsely populated might lead to the conclusion that it be more efficiently implemented as a hashed list. Determining density of arrays is a good deal more difficult than identifying the maximum subscript. In all events, table size must include the effects of preset data, which further complicates the problem.

3.2.1.3 Compiler Performance Profiles

The report describes the kinds of desirable compiler performance data in order to obtain an objective basis for compiler efficiency improvement. The most critical measure - assuming that real performance is not attempted until the compiler error/failure rate is reduced to very near zero - is the execution profile; i.e., where is the majority of compile time spent and what are the data requirements of the largest phase? Of course, obtaining compiler execution profile data is not unlike that of obtaining user program profile data; the same artifacts may be used, especially if the compiler is written in its own language and self-compiled. For example, upon discovering that a significantly large percentage of compile time is spent in a particular routine, the routine may then be subject to various kinds of tuning and manual optimization (including perhaps rewriting in machine code).

The report seems to imply that a typical compiler often spends a large percentage of its execution time in parsing and backtracking. However, this is not indicative of all compilers. In fact, a compiler that spends 35 percent of its execution in its parsing and backtracking may be a good candidate for redesign without further investigation. By itself, such a figure is not very useful; perhaps the compiler does no optimization or only primitive code generation in which case 35 percent in parsing is probably irreducible.

The report provides some useful curves of compiler performance:

- Compilation time per statement as a function of source program size
- Compilation time per statement as a function of statement size
- Size of object program generated as a function of source program size

These curves are extremely interesting; however, responding to them may be difficult and costly. A curve that shows decreasing compile speed with program size probably represents a fundamental design problem that perhaps only a new compiler can correct. Of course, that information is useful in itself.

The following data in graphic form would be useful supplements to the list:

- Core occupancy of the compiler as a function of source program size
- Core occupancy of the compiler as a function of compiler options (listings, optimization, etc.)
- Compilation speed per statement as a function of selected options
- Size and number of scratch files generated as a function of source program size and options

3.2.2 DATA ANALYSIS

For the gathered data to be useful, it must be collected, subjected to various statistical processes, and displayed. The report asserts that at this point, the analyst's intuition comes strongly into play; once the numbers are before him, his insights are tempered by the figures, resulting in a more sensible analysis. In Chapter 3, various kinds of displays are suggested. To aid in the analytical process, the report suggests many kinds of displays of reduced data. The most useful are:

- Histograms of static language feature usage
- Histograms of dynamic language feature usage
- Histograms of functional segmentation of program execution
- Plots of error persistence
- Plots of compile time per statement versus statement length, source program size, etc.

The report also considers a little-explored aspect of language evaluation: cost/performance ratios of various language forms. Similarly, examination of statistics may lead to an objective measurement of the suitability of a language to its application. The report suggests that the following kinds of computed ratios are useful in this analysis:

- Development cost versus frequency of usage for language features
- Ratio of errors to usage by language feature
- Ratio of errors to program size
- Ratio of error persistence to program size

The report discusses the difficult area of program transferability. It is suggested that a proper analysis of the data may lead to an inference about program transferability. It is further pointed out that the heavy use of DEFINE parameters and a low incidence of DIRECT code may be correlated with more ready transferability. The report clearly warns that the measurement for transferability is uncertain at best; CSC concurs. Programmers often rely, perhaps quite unwittingly, on characteristics of the target-machine hardware.

Obviously, defined entry tables are disastrous to portability. Less obviously, reliance on unary minus to yield a logical complement is similarly unfortunate. Parameterizing number precision is good theoretically, but is extremely difficult to realize without resorting to multi-precision interpretive arithmetic to guarantee machine independence. Programmers are as apt to be guided by the objective of program efficiency as by portability, which results in many hidden impediments to transferability.

An analytic tool (described in Chapter 3 of the report) that seems to hold considerable promise is the appending of an execution histogram to a source listing of the program to be measured. This is of importance in analyzing program or system performance, to find unused code, and to identify and then tune program bottlenecks. Creating such a display is a difficult task and would require a thoughtful integrated design approach that the authors of the report do not address further. This problem is addressed in subsequent sections in this study.

3.2.3 DATA COLLECTION

3.2.3.1 Hardware Instrumentation

The report discusses the use of hardware instrumentation to achieve event counting and program sampling. This is clearly the most difficult approach to data gathering for several reasons:

- Cost of equipment
- Cost of setup for measurement
- Reliance upon implicit knowledge of system organization in terms of table formats and addresses of critical data and routines
- Lack of transferability of techniques to other machines

The advantage of hardware instrumentation is primarily to reduce the cost of obtaining the data, since it offers the least perturbation of the system performance while measurement is in process.

3.2.3.2 Preprocessor Approach

The report discusses at length a technique of building a preprocessor to analyze the source JOVIAL program, gather the static data, and instrument the source by modifying it in order to generate calls to data gathering routines during execution

Another idea mentioned is to intercept command invocation of the JOVIAL compiler and enter the preprocessor instead, then call the compiler with the modified (instrumented) source. This approach seems reasonable since inserting these same functions into a JOVIAL compiler so large and sensitive to tampering really is impossible. This approach cannot be completely rejected: but instead of intercepting the compiler invocation, CSC suggests integrating the preprocessor with the compiler under a single executive with a control program to manage the sequence of execution. This method would achieve the desired end without disturbing existing code.

3.2.3.3 Super Compiler Approach

The alternative to the preprocessor approach is to modify the existing JOVIAL compiler to perform the same functions. The JLMT report calls this modified compiler a super compiler. This approach carries the obvious advantage of eliminating duplication of work. Thus, the preprocessor must perform many of the same functions as does the compiler front-end, and the preprocessor cannot gather certain important compilation statistics which still would have to be obtained by modifying the compiler itself.

If a standard compiler is to be implemented for the LCF, then part of this standard should be the statistics gathering function. This objective is then more directly achieved by incorporating the statistics gathering functions directly into the standard JOVIAL compiler. Because this allows rehostability of this element of the measurement tool, CSC greatly favors this approach.

3.2.3.4 Software Instrumentation

The suggested method of instrumentation is to modify the source code to insert calls to measurement subroutines. This is the cheapest approach to implement, but it does present some difficulties, such as the necessity to insert BEGIN-END brackets where they were not originally required. For example, the statement:

```
IF cond; XX = XX + 1
```

is instrumented to record the execution of the assignment statement. Clearly,

```
IF cond; MEASURE(. . .); XX = XX + 1
```

has altered the meaning of the program. The required bracketing is:

```
IF cond; BEGIN MEASURE(. . .); XX = XX + 1; END
```

Another disadvantage to source language instrumentation is that great numbers of parameters must be passed in order to achieve the collection of the kinds of data suggested in the report.

An alternative to the source language insertion technique (not discussed by the report) is to build the instrumentation into the intermediate language. This is the approach used in most compilers to effect the MONITOR statement. While in itself this does not reduce the requirement for parameters, a more integrated approach to instrumentation does. For example, the compiler may build a statement descriptor table for each statement which contains encoded forms of the necessary parameters. Using this approach, the call to the measurement routine needs only a single parameter to identify the statement descriptor table entry. The measurement routine call might not have to be made. Instead, execution of the program proceeds in a partially interpretive mode with a control monitor obeying the sequence described by the statement descriptor table and collecting the data during the interpretation. Although interpretation is usually avoided, the insertion of measurement monitoring as alterations to the source has much the same effect in degrading program performance and bears the additional cost of greatly enlarging both the source and object programs (perhaps beyond manageable limits) thus defeating the approach.

The report addresses the problem of data recording only superficially. The problem of data recording must be considered as part of an integrated statistics gathering design. The suggested approach of generating in-core tables of data to survive from compilation through link-editing and into execution is too delicate to be practical. It requires each of the processors (including the data analysis and reduction programs) to have inherent knowledge of fixed locations for the data and, of course, eliminates the possibility of separate jobs for compilation, link-editing, execution, statistics gathering, and analysis.

A more modular approach, which still requires modifications to the compilers and link-editors but leaves the programs largely independent, is to define file *structure conventions for the storage of the data.* (See Section 8 of this volume for a suggested approach.) Statement descriptor files and static statistics files can be produced as a compilation output in much the same way that relocatable binary files are output. Similarly, the link-editor creates composite statement descriptor *tables and statistics tables, which may be written to files.* When executing in the statistics gathering mode, the interpreter (or called library routines) may generate additional files. Finally, the data reduction programs should have access to these files to collate, summarize, and display for analysis.

Some consideration should be made for choosing either file replacement or file appending as the normal mode. Statement descriptor tables clearly replace any earlier version, but compilation statistics may be appended more properly to earlier statistics for the same program (the approach may be to use the IBM-style generation files for this purpose). For example, the "append" option allows analysis of items such as error decay with program life.

For maximum utility to programmers, language designers, project supervisors and installation managers, the method of data recording and the convenience of its subsequent retrieval for display and analysis is probably the most important design consideration. The JLMT report treats this subject minimally.

3.3 CONCLUSIONS

The report succeeds in identifying useful categories of statistics to be gathered. It is a helpful document in defining broad requirements in the area of language measurement. It does not, however, progress to the point of defining functional specifications for a viable system. In particular, the notes on implementation consist more of speculation than concrete proposals and too much detail is presented on the preprocessor approach. The most important shortcoming is the failure to specify a rational, integrated approach to data recording with a description of the interfaces among the principal elements of the measurement system. Indeed, the elements themselves are not clearly identified. A language measurement tool would satisfy a real need in the LCF. In Section 11 (statistics collection for the LCF) the omissions noted here are addressed and corrective measures are recommended.

SECTION 4 - JOVIAL COMPILER VALIDATION SYSTEM (JCVS)

4.1 DESCRIPTION

The JOVIAL Compiler Validation System (JCVS) was implemented to provide a standard method for verifying that all JOVIAL compilers implemented for the Air Force meet the requirements as set forth in the JOVIAL standard defined in AFM 100-24. Two such systems have actually been implemented: one for JOVIAL/J-3 and another for JOVIAL/J73 language and its subsets.

The J-3 JCVS has been used to measure most existing J-3 compilers on several different machines, including the JOCIT J-3 compiler for the HIS-6000/GCOS system. The J73 JCVS has been used to validate the J73/I compiler implemented for the DECSYSTEM-10.

The JCVS is constructed as a set of JOVIAL source language statements organized into functional modules, each designed to exercise a specific feature of the language. They are divided into six classes:

- Class 1 - Language Features
- Class 2 - Error Detection and Reporting
- Class 3 - Capacity
- Class 4 - Efficiency
- Class 5 - Machine/Implementation Features
- Class 6 - Generalized/Other Tests

The tests are constructed systematically to measure each feature independently in an exhaustive set of variations. The tests are organized with a large set of declarations common to all the tests. In addition, target machine characteristics are parameterized, and certain subroutines that generate printed results and analyze success and failure are provided as common routines. The tests are further annotated to a considerable degree in the form of source language comments. A manual describing the tests and their use provides the documentation that supports each of the JCVS sets.

The tests are constructed so as to be entirely self-checking. They are designed so that a commentary briefly describing the test is printed before an announcement of success or failure. The output of each test further identifies the module to provide a cross-reference both to the JCVS source listing and also to the documentation describing the tests. The common declarations and routines and the individual modules are maintained in a large source text library.

It was the original intent in the design to provide a test construction facility in which the test preparer would identify the modules to be selected, and a complete test program - in the form of a single source module - would be generated automatically. This design was realized for the J-3 JCVS but was not realized for the J73 set. The J-3 test constructor was written in COBOL, and was implemented under GCOS; the J73 test constructor was not actually implemented. To some degree, the test construction facility may be duplicated, although with less convenience, with the services of standard text editing facilities.

The tests for both J-3 and J73 consist of several thousand lines of source code and are quite comprehensive. Both the JOCIT J-3 compiler and the DECSYSTEM-10 J73/I compiler underwent an exhaustive test cycle of self-checking tests prepared by the contractor. Upon completion of this level of system testing, final testing and acceptance of the compilers consisted of successfully exercising the JCVS tests. An additional level of validation was achieved in the J73/I compiler by writing it in its own language; self-compilation was included as a further measure of successful implementation.

4.2 ANALYSIS

The principal value of the JCVS tests is that they have been prepared entirely independently of any compiler implementation activity. They have been written to exercise the language constructs as defined in the language standard specifications. This adds a level of objectivity to the validation/acceptance activity that is not readily obtainable when the testing is left solely to the compiler implementers. Systematic testing by the implementers is not precluded by the

JCVS tests. On the contrary, the contractor's own testing activity is designed to bring the compiler to a state of readiness, after which the application of the standard JCVS tests serve to certify it for release and production service. Another of the virtues of the JCVS tests is that the tests are organized into functional isolation, and failures that they manifest are readily identifiable, thus shortening the analysis and correction cycle considerably. The tests are quite disciplined in avoiding inter-module dependencies that would make the analysis more complex.

Although the JCVS tests have been designed to test the JOVIAL language exclusively, the principles of test development are readily adaptable to any other language. The JCVS concept was judged to be outstandingly cost-effective and a generally excellent tool for measuring the success of compiler implementation efforts. While the existing JCVS tests are JOVIAL-based, it is realistic to expect that much of the tests would submit to language translation for adoption to non-JOVIAL HOLs. Of course, the translated tests would have to be supplemented by additional modules to measure language features not part of the JOVIAL languages, and certain modules would be deleted as inapplicable. Nonetheless, the guidance offered by the existing tests should aid significantly in reducing the effort to prepare validation measurement for other languages.

Both the J-3 and J73 JCVS tests exhibit some shortcomings. Correcting these would further enhance the suitability of the tests in supporting a JOVIAL-based LCF. These shortcomings are both minor (in general easy to remedy) and major (correction would require significant effort).

The minor deficiencies primarily result from insufficient testing of the following features:

- Input/output
- ENCODE/DECODE
- MONITOR/TRACE
- BIT, BYTE, and ENTRY assignments and relationals
- Optimization

The major deficiencies may be further divided into those that result from a lack of testing and those which are simply procedural. One area, not tested adequately by JCVS but very important to the normal user of JOVIAL, consists of those aspects of the language and compiler related to system construction and integration. In particular, the tests fail to exercise the COMPOOL capability for describing external data, initialization of COMPOOL data, declaration of external procedures/data and their attributes; the capability to generate linkable, separately compiled modules; and the interface with other system programs and libraries. Undiscovered errors in these areas are left to the unsuspecting user to find. A similar area is left for the user to detect results from feature interaction. Although the JCVS tests perform an excellent validation of independent features, the difficult and lengthy phase of compiler verification is that which involves detection and correction of errors as language constructs interrelate. Some brief examples follow:

- Procedure calls may work independently, and formal parameter usage may work independently, but the call mechanism may fail when formal parameters are passed as actual parameters
- Parallel and serial tables may work independently but fail when used together
- Recursion with dynamic storage allocation is always a menace
- Optimization is often thwarted by overlaid data, COMPOOL data, and improperly analyzed program flow
- Initialization of compiler-packed and like tables

This form of testing is complex and is best performed by real programs. Compilations of the compiler itself have been an excellent method to discover errors resulting from feature interaction, but not all compilers are self-compiled (JOCIT, for example), and additional tests which are organized especially to test these interactions should be designed and included.

The procedural deficiency lies in the absence of a cataloging index to the tests and test modules by feature. With such a catalog and an automated means of assembling the desired modules, the test could prove to be much more useful for phased testing during compiler development. The J73 JCVS assists somewhat in providing appendixes to the user's manual to cross-reference language features and test modules.

4.3 CONCLUSIONS

With some enhancements, discussed in Section 12, the JCVS tests for both J73 and J-3 provide a useful basis for compiler validation in an LCF supporting these languages. The automated test construction facility should be strengthened to give greater convenience to both the development testing activity and also the acceptance testing activity. The JCVS concept is an equally useful base from which to develop validators for other HOLs.

SECTION 5 - JOVIAL AUTOMATED VERIFICATION SYSTEM (JAVS)

5.1 DESCRIPTION

The JOVIAL Automated Verification System (JAVS) is a tool designed to assist in developing programs written in JOVIAL/J-3. The primary purpose of JAVS is to aid the programmer in testing. JAVS is a program that operates on JOVIAL/J-3 source programs. The programmer may employ JAVS when his programs have compiled successfully without error and have demonstrated that they may be executed through a minimal path successfully. At that point JAVS can provide the following services:

- Analyze and format the source text
- Perform flow analysis
- Insert instrumentation for performance measurement
- Describe inter-module relationships
- Generate test measurement results

JAVS is described in two documents: the User's Guide and the Reference Manual. The program operates on the HIS-6000 under GCOS. It is divided into six separate functions known as STEPs, which are invoked by means of a control language described in the documentation. JAVS operates on JOVIAL/J-3 source modules and requires the services of the JOCIT JOVIAL compiler to create executable modules. JAVS assumes certain implicit knowledge of the particular J-3 language processed by JOCIT; namely, it employs the MONITOR statement, as explained in paragraph 5.1.3, Instrumentation. The functions provided by JAVS are described in the following paragraphs.

5.1.1 ANALYZE AND FORMAT SOURCE TEXT

This process is known as STEP 1 in the JAVS system. During STEP 1, the JOVIAL program is read, and various tables are created for subsequent processing by later steps. In addition, the user has the option to produce various kinds of listings of his program with supplemental annotation. This step is similar to the syntax analysis phase of a compiler, and in fact, it duplicates many of the functions of such a phase. JAVS diagnoses incorrect programs to some degree, and advises the user when certain JAVS limitations and capacities are violated. (One such limitation is that character variables may not exceed 120 characters.)

The programmer directs JAVS partially through the inclusion of special comments in the source. These are ignored by the J-3 compiler, but are recognized by the JAVS program. Such information as module identification, or COMPOOLs to be referenced are inserted by the programmer.

A large variety of listings may be obtained from the JAVS system. The most useful of these show the source text reformatted with indentation for BEGIN-END blocks, JAVS-assigned statement numbers, etc. There is also the option to show expanded DEFINE name references. Many of the supplemental listings (such as the statement descriptor table) are intended for JAVS maintenance, and are of marginal relevance to the programmer's testing effort. The primary output of STEP 1 is the module descriptor table required by subsequent steps.

5.1.2 FLOW ANALYSIS

Flow analysis is described in JAVS terminology in terms of decision-to-decision paths (DD-paths). Basically, a DD-path is the code that occurs between one conditional branch and the next. The flow analysis is performed by STEP 2 and generates tables (into the JAVS library) for use by subsequent steps. The flow analysis is expressed in the form of descriptor tables which may be processed by JAVS report-writer functions to generate annotated listings of the program text.

STEP 4 is also used to provide flow analysis information. Its output is entirely in the form of detailed reports, any one or all of which are selectable during STEP 4 processing. Detailed analyses of DD-paths are available in several forms including an annotated source listing showing just the source text for the decisions, a stylized graphlike listing showing the reach of each DD-path and, of considerable interest, the summary listing of each statement by type (procedure, assignment, IF, etc.) and the DD-paths containing the statement. STEP 4 also provides the facility to generate a cross-reference list of the modules contained in the user's library. This function duplicates that of the JOCIT compiler but provides a composite listing while omitting such data as variable type, scope, etc., which are provided by the JOCIT compiler.

5.1.3 INSTRUMENTATION

STEP 3 performs instrumentation of the source module. STEPs 1 and 2 will have been run before STEP 3. The steps are not sequenced automatically as are the phases of a compiler; the step selection and ordering are controlled by the programmer. If something is wrong or out of proper sequence, JAVS gives notice. Instrumentation is implemented through the use of JAVS directives, which are special JOVIAL comments. Unfortunately, the JAVS documentation does not describe the syntax of the special comment; one is left to infer that what makes them special is the occurrence of a dot (".") as the first character of the comment. The effect of blanks is not discussed, and in general, the user is not advised or cautioned about how his program commentary might unwittingly be processed by JAVS. There are three main forms of instrumentation: flow tracing, variable assignment tracing, and value verification.

Flow tracing includes subroutine entrance/exit tracing, and DD-path tracing. Asking for these causes JAVS to insert JOVIAL statements that call predefined subroutines (in the required special JAVS COMPOOL) that build JAVS tables during execution to record the trace data. These tables are then processable by elements of the JAVS system (STEP 6) after execution for analysis.

Variable assignment tracing is implemented by making use of the JOCIT compiler MONITOR feature. Asking that a variable be traced causes JAVS to insert an appropriate MONITOR statement. Such tracing may also be turned off by including a JAVS-generated Boolean variable as the conditional expression in the

MONITOR statement. These JAVS-generated variable names may conflict with programmer names; the programmer discovers such duplication only after compiling the program with the J-3 compiler. Value assignment tracing is not inserted into the JAVS files for subsequent analysis; instead, it appears as part of the standard print file output. Writing the trace directive, however, is somewhat simpler than the corresponding MONITOR statement if both ON and OFF trace functions are desired: if only the ON function is desired, the MONITOR statement is equally as useful. Of course, the JAVS directives, appearing as comments, do not generate code when the original source text is compiled without JAVS pre-processing.

Value verification is useful to the programmer. It allows him to specify - again, in the form of a directive - what values, or range of values, a variable is expected to contain at selected points in the program. (The selected points are those at which the directives appear.) These directives, known as ASSERT and EXPECT, also generate MONITOR statements that produce output only when the assertion is false or the expectation is not fulfilled. The output goes only to the print file, similar to the value tracing; JAVS does not intercept the information for its own purposes.

The output of the instrumentation step is a new J-3 source program with the inserted subroutine calls and MONITOR statements included. This program must be compiled by the JOVIAL compiler as a step separate from and independent of the JAVS system.

5.1.4 INTERMODULE RELATIONSHIPS

JAVS is usefully oriented to verification of systems employing multiple modules. STEP 5 of the JAVS system generates reports and listings that describe the called and calling relationships among programs in the user's set and with the library as well. The only limitation to the usefulness of this step is the number of modules that can be named (vertically) across the printer page (115 is the

JAVS limit). One report, which appears to be most useful in integrating multiple-module systems, is the hierarchical module diagram. Unfortunately, only ten levels are supported, which seems inadequate for describing real, well-structured programs.

5.1.5 TEST MEASUREMENT

After a JOVIAL program has been processed through STEPs 1-5 and then compiled and executed, STEP 6 of the JAVS system is employed to assist in post-mortem analysis. Instrumented flow-tracing data will have been recorded by the JAVS object-time subroutines in the JAVS library. This data is not retrievable by STEP 6 and may be formatted into any of several reports.

Of primary interest to the programmer is to observe how much of his program has not been tested. JAVS usefully segregates its trace output by separate executions of the instrumented program, such that statistics for any particular run or cumulative statistics for all runs are available. The cumulative statistics show, for example, which DD-paths were not exercised. Such information may be reported in tabular form or as an annotation to a reproduced listing of the original source text. This is an excellent concept, and it cannot be difficult for a programmer to use such an annotated listing and quickly identify the DD-paths that were not exercised, and relatively quickly (depending upon program complexity) generate another test or set of tests to cover the unexercised paths. Useless code as well as improperly written conditional statements may be discovered by such an analysis.

In addition to these summaries, STEP 6 also provides a step-by-step trace of each DD-path, showing the branch condition or switch path taken. Finally, execution-time summaries by module are available, although it is not known to what degree these timings are skewed by the timer function itself, implanted instrumentation, etc. Very little is said about the timing summary; this part of the manual should be augmented because timing is usually critical in real systems, and effective measurement would aid in their development.

5.2 ANALYSIS

JAVS has been developed expressly for the JOCIT/J-3 language as implemented on the HIS-6000 machine under GCOS. Oriented to that language, JAVS appears not to be directly transferable for use with other languages. Although the documentation does not mention it, discussions confirm that JAVS is implemented only for the JOVIAL/J-3 language. Therefore, CSC concludes that it would require substantial effort to redevelop JAVS for some new language.

However, some aspects of the design do have readily portable characteristics, such as module instrumentation. For example, the flow tracing, ASSERT, and EXPECT directives could be implemented somewhat similarly for J73 through use of the !TRACE directive. The JAVS architecture and file interface design also appears to accommodate multiple languages. What is not known is the degree to which the internal tables are specifically oriented to J-3. By inference, there is some orientation because of the way statements are classified (for example, "IFEI" or "GTCL", which have no direct counterparts in other languages), and this would demand some retailoring for new languages. As such, JAVS may not be considered in the category of language-independent tools. However, the concepts are valid for other languages; especially in the area of post-mortem summary analysis, JAVS points the way to a useful program development aid in the LCF.

The major advantage of a JAVS-like tool is its application to multimodule systems. Large programming systems are invariably complex in organization, and as structured programming techniques become more widely applied, the ability to deal with multiple modules will be a major requirement at all points in the program development spectrum. As described in the documentation, JAVS addresses the problem, but perhaps falls short in its capacities. Real systems may be expected to contain several hundreds (perhaps thousands) of modules, and it is not clear from the JAVS description whether or not the table capacities may be expanded to meet such a need. Furthermore, the documentation does not make clear exactly what a module is. By inference, a module must be a source unit of code that defines either a main program or an external procedure, both of which may contain any number of internal procedures. If JAVS assumes each procedure to be a module, then the stated capacities present an even more serious restriction.

The control language for driving JAVS is somewhat verbose, and the division into steps seems more oriented towards the JAVS program architecture than to user convenience. STEPs 2 and 4, for example, are not meaningfully distinct from the user's functional viewpoint, and perhaps could be consolidated to make the user interface simpler. In fact, requiring the user to divide processing into JAVS steps at all seems unnecessary. What the user wants is (1) to generate annotated listings for flow (DD-path) identification, (2) to produce formatted listings (if the compiler cannot do it for him), (3) to instrument his program for execution measurement. and (4) to have access to postmortem analysis facilities. It seems that a much simpler and more direct control language - not STEP-oriented - would fulfill these basic requirements in a more useful way.

A point of concern in measuring large programs is that JAVS instrumentation may be expected to add considerable volume to the generated code. The procedure calls for flow-tracing and the MONITOR statements consume both compile-time and run-time resources that may make measurement difficult, if not impossible. Of course, a stand-alone JAVS can hardly avoid the choice of source-language instrumentation. The only alternative is for the compiler to provide some encoded output that is processed by a run-time monitor to insert the probes at the machine language level. However, this approach defines an entirely different discipline not attempted by the JAVS developers. This problem coupled with the capacities problems restricts JAVS applicability to large program development.

5.3 CONCLUSIONS

As implemented, JAVS is relevant only in a J-3 LCF. It is quite useful in verifying and developing tests for medium scale programs. Its strongest feature is the postmortem analysis aids, which include both selected run and cumulative execution data. Users of JAVS may find that instrumented programs are considerably larger than the uninstrumented ones, and therefore performance measurement results may be skewed beyond usefulness. The performance effect may be compounded further by the interaction of probe and monitor code with the application of optimizing algorithms by the compiler. (For example, MONITOR calls may destroy values held in registers and force temporary stores and reloads not required in the uninstrumented program.) However, JAVS is a useful companion to the JOCIT J-3 compiler within the scope of a J-3 LCF, and the JAVS principles and functional design may be fruitfully applied to other HOLs.

The JAVS tool, implemented as a preprocessor, presents the problem of ensuring and maintaining compiler and JAVS processor language compatibility. A more economic approach would incorporate the analysis and instrumentation functions (STEPs 1-5) into the compiler. The slight increase in compiler development cost and size and the negligible loss in performance are more than offset by the beneficial elimination of a redundant language processor, the elimination of language incompatibilities, significant improvement in instrumentation efficiency, and the guarantee of portability.

not make clear exactly what a module is. By inference, a module must be a source unit of code that defines either a main program or an external procedure, both of which may contain any number of internal procedures. If JAVS assumes each procedure to be a module, then the stated capacities present an even more serious restriction.

The control language for driving JAVS is somewhat verbose, and the division into steps seems more oriented towards the JAVS program architecture than to user convenience. STEPs 2 and 4, for example, are not meaningfully distinct from the user's functional viewpoint, and perhaps could be consolidated to make the user interface simpler. In fact, requiring the user to divide processing into JAVS steps at all seems unnecessary. What the user wants is (1) to generate annotated listings for flow (DD-path) identification, (2) to produce formatted listings (if the compiler cannot do it for him), (3) to instrument his program for execution measurement, and (4) to have access to postmortem analysis facilities. It seems that a much simpler and more direct control language - not STEP-oriented - would fulfill these basic requirements in a more useful way.

A point of concern in measuring large programs is that JAVS instrumentation may be expected to add considerable volume to the generated code. The procedure calls for flow-tracing and the MONITOR statements consume both compile-time and run-time resources that may make measurement difficult, if not impossible. Of course, a stand-alone JAVS can hardly avoid the choice of source-language instrumentation. The only alternative is for the compiler to provide some encoded output that is processed by a run-time monitor to insert the probes at the machine language level. However, this approach defines an entirely different discipline not attempted by the JAVS developers. This problem coupled with the capacities problems restricts JAVS applicability to large program development.

SECTION 6 - SEMANOL

6.1 DESCRIPTION

SEMANOL (Semantics-Oriented Language) is a language and a programming system whose purpose is to describe the syntax and the semantics of certain classes of languages. SEMANOL was developed with two principal objectives in mind:

- A rigid methodology that forces the unambiguous and machine-verifiable statement of the total semantic effect of the language under definition
- A resulting definition comprehensible to the reader as a language specification standard

6.1.1 ORGANIZATION OF A SEMANOL DESCRIPTION

A full SEMANOL description looks very much like a program written in a high-order language. The description consists of four main parts:

1. Context-free syntax description
2. Noncontext-free syntax description
3. Semantic description
4. Control section

Part 1 of the description uses a generative grammar much like the Backus-Naur Form (BNF). The grammar is written as a set of syntactic definitions, each of which takes the form of a "production" that defines a single syntactic construct in terms of other constructs and/or terminal symbols. The definitions may be recursive in that the construct being defined may itself be an element of the definition, directly or indirectly. The total context-free description may be viewed as a tree with a principal construct - say, <program> - at the root node, and the set of terminal symbols forming the lowest-level branch termini. The context-free part of the SEMANOL description is the only one of the four parts that does not have a procedure-oriented appearance.

Part 1, then, forms the total syntactic definition, while Parts 2, 3, and 4 consist of a highly recursive set of procedures and functions which execute as a sequential machine to mechanize the semantic description. The operation of the SEMANOL machine is described at the highest level by Part 4, the control section. Parts 2 and 3 may be thought of as a set of subroutines of Part 4.

The input to SEMANOL is a particular program written in the language defined by the four part description. First, the control section causes a parse tree of the particular program to be constructed using the Part 1 grammar. Next, the noncontext-free analysis (recognizing undefined names, etc.) is performed by Part 2. Finally, the semantic effects of the particular program, together with its inputs, are determined by applying the appropriate semantic description from Part 3 to the nodes of the parse tree in particular-program-order; i.e., the "execution" of the particular program is simulated in an interpretive manner.

6.1.2 THE SEMANOL SYSTEM

The SEMANOL system is organized into two major components: the translator and the interpreter. The translator operates on the four part description (described in the paragraphs above) and creates an encoded intermediate representation known as SIL (SEMANOL Internal Language). The SIL is then given to the interpreter, which performs the execution. This execution produces outputs and constitutes the machine verification of the language description. The thoroughness of the verification, of course, depends upon the comprehensiveness of the particular program or programs which may be thought of as a set of test cases for the language.

The SEMANOL system is written in FORTRAN and operates under the MULTICS system. Execution of SEMANOL is not unlike that of a compiler, except that the execution, being interpretive, is quite slow. Because the system is coded in FORTRAN, it is alleged to be relatively portable. In fact, the first implementation of SEMANOL was begun under GCOS and later rehosted to MULTICS. However, because significant changes were incorporated then, a true measure of rehostability was not achieved.

6.2 ANALYSIS

The study of SEMANOL and the conclusions drawn about it were mostly based on several existing documents that formally describe both the SEMANOL system, and two separate language specifications (one for JOVIAL/J-3 and one for JOVIAL/J73). In addition, the authors of the system gave a presentation on SEMANOL and answered inquiries during the course of the LCF study. The SEMANOL system discussed here is the second-generation system, known as SEMANOL (73), which (1) incorporates changes to correct shortcomings in the earlier version and (2) specifically supports the writing of the JOVIAL/J73 specification.

The J73 specification itself stands as a useful example for study purposes, and it is greatly strengthened by the frequent use of English commentary. An attempt was made to gain hands-on experience with SEMANOL by exercising the J73 specification. However, this attempt was defeated by the fact that SEMANOL had become temporarily inoperative because of recent modifications to MULTICS, which changed the meaning of certain FORTRAN I/O statements in the SEMANOL program. It was hoped for a time that the developer would be able to bring up SEMANOL again during the course of the LCF study, but lack of funding prevented it. It is unfortunate, because direct experience invariably lends greater dimension and objectivity.

Within the framework of an R&D effort, the SEMANOL project has the appearance of being successful. The documents are quite readable, although a lack of a tutorial text limits the readership to experienced language specialists. The authors are clearly knowledgeable and given to very clear expression about their system. The system they have created appears to be a well-conceived tool.

6.2.1 THE CASE FOR LANGUAGE SPECIFICATION

The evaluation of SEMANOL presented in this section must be understood in the context of the requirements for language specification within the LCF. Clearly, the very first step in providing for meaningful language control is to establish an unequivocal definition of the language in question. Such a definition serves to:

- Provide a basis for a programming standard for the language users
- Provide clear and unambiguous guidance for compiler implementers
- Create a standard that ensures program portability among the possible target machines within the LCF community
- Stifle the proliferation of non-standard implementations

The primary issue is the method of achieving these goals in the most cost-effective manner. There are several conflicting objectives; it is not clear that a single form serves each equally well. For example, a formal description that guides a compiler writer is not likely to serve the needs of a less-experienced applications programmer. The former requires a formal, rigorous specification of the language while the latter needs a well-exemplified, tutorial treatment.

One way to approach a definition is to build a compiler implementation tool that is truly independent of the target machine. Then the compiler tool becomes the standard. But there are rational objections to this approach. For one thing, compilers are difficult to read. The semantic effects are distributed throughout the compiler phases, the library, and even the link editor and operating system services. Some argument can be made that once an implementation standard is established through such a tool, there is no need for a rigorous specification since all implementations are defined by the tool. However, this fails to provide guidance for implementations of the language outside the community and certainly fails to provide language users with anything meaningful or readable. Instead, an implementation standard should be derived from a higher specification that is subject to analysis and review outside of, and in advance of, any implementation activity.

In the case of JOVIAL J-3, several implementations failed to achieve a consistent interpretation. Part of the blame for this can be placed on the inconsistencies and incompleteness of the language standard defined in AFM 100-24, but AFM 100-24 cannot bear the sole responsibility; it can even be criticized for overspecification in certain cases.

There has been much learned about language design and description since the days of the JOVIAL definition. What is sometimes criticized as a poor language specification for JOVIAL is often more accurately perceived as poor language design. Early vintage HOLs have many characteristics that invite nonstandard usage, that permit ill-advised access to target machine characteristics, and that otherwise defeat program portability. It is CSC's feeling that no amount of careful specification of these languages would significantly ease this condition. For example, the current mainstream of language design insists on inviolable compile-time type checking (e.g., the DOD "TINMAN" specification, ALGOL-68, PASCAL) to which such features as FORTRAN EQUIVALENCE, and JOVIAL specified tables stand in fundamental contradiction, and which features defy the kind of universal semantic specification that meets the previously stated objectives.

CSC agrees completely with RADC's intent in providing a complete and unambiguous language specification for the HOL or HOLs supported within the LCF. However, CSC questions whether the search for tools to support this goal will help correct the more fundamental problems with the design inherent in the languages which RADC might realistically consider for the first LCF implementation. This is not to say that existing specifications do not merit some concerted effort to refine their description and at the very least to identify the three principal categories of sub-specification, namely, the universally defined, the undefined-by-this-description, and the implementation-defined.

Earlier, CSC rejected the idea of using SEMANOL in the LCF; but that judgement was largely based on expectations as to which HOLs were the likeliest candidates for LCF support. These languages (largely, of the JOVIAL family) simply have such a dense weighting of the undefined or implementation-defined, that CSC concluded that the application of a SEMANOL approach to language specification was not cost-effective. The poor cost-effectiveness is largely due to the necessity to rewrite the implementation-defined semantics for each target and the necessity to rationalize the undefined such that implementations would not be judged incorrect because their realization of the undefined differed from the SEMANOL realization. However, in the next generation of languages the universal definition overwhelms the undefined and implementation-defined, and the implementation-defined semantics are at least parameterized to a high degree through language constructs. Consequently, SEMANOL will serve as a useful companion tool to the language definers during language definition.

As the following analysis shows, SEMANOL does not serve all the previously stated objectives equally well. What is more urgently needed is a clear and complete specification for implementers and an equally clear but more tutorial description for language users, neither of which SEMANOL fully meets. Several forms of language specification emerging in the last few years hint that these objectives may be met. Not all are equally successful, but it is difficult to judge adequately because they have introduced a new technology and vocabulary which are unfamiliar and often misunderstood. The ALGOL-68 description, for example, in which the two-level syntax has been employed to include the context-sensitive portions of syntax specification (e.g., mating of defining and applied occurrences of identifiers, detection of equivalent modes, etc.) is a significant achievement, but it suffers from an impenetrability due almost as much to the complexity and difficulty of the language itself as to the descriptive methodology.

The ALGOL-60 description has provided the model for nearly all subsequent specifications, and attempts to refine it have not always resulted in an improvement. The description of PASCAL[1] combines a highly refined language with one of the most transparent specifications ever written. Based on a BNF-style formal syntax description and supplemented by semantic footnotes and numerous examples, the PASCAL Report is a good model for a language description. However, it is not certain whether the clarity and completeness of the Report influences an assessment of the language or, conversely, whether the simplicity of the language is actually responsible for the clear specification.

A final point to be made is that any language specification describes an abstract machine (Turing machine, Chomsky Type I machine, PASCAL machine, SEMANOL machine). The problem is to find the machine which is intelligible to the reader and which can also be "translated" to a set of real machines using some readily-derived, minimal set of mapping functions. At this point, only subjective judgements can be made; however, within an R&D environment, the technology is best served by experimenting with different forms, one of which is the SEMANOL form. The following paragraphs examine some of the characteristics of SEMANOL as measured against the needs of the LCF.

6.2.2 OVERSPECIFICATION

The desirability and meaning of an unambiguous specification is open to question. Certain English-language descriptions have in the past been criticized for overspecification. For example, JOVIAL J-3 has specified clearly that integers are to be represented in signed-magnitude format. This specification, if followed absolutely, would have crippled the implementation effort for all compilers but those intended for the IBM-7000 series of machines. No JOVIAL compiler, except that for the signed-magnitude 7094, has ever adhered to this specification.

---

[1] Jensen & Wirth, "PASCAL User Manual and Report," Springer-Verlag, Berlin 1974.

Hoare and Wirth observed: "Since one of the principal aims in designing PASCAL was to construct a basis for truly portable software, it is mandatory to ensure full compatibility among implementations. To this end, implementers must be able to rely on a rigorous definition of the language. The definition must clearly state the rules that are considered binding, and on the other hand give the implementer enough freedom to achieve efficiency by leaving certain less important aspects undefined.[2] This seems to pinpoint the problem: to identify "the important aspects." It can be argued that, for example, the effect of assigning a value to a variable whose declared range does not include the value should be undefined. In this case, an optimizer can take considerable advantage of this in recognizing redundant computations by the so-called "value-folding" technique. It can also be argued that such a lack of definition leads to the construction of wrong programs in which errors arising from undetected truncation are very hard to isolate. Another, more difficult point, is in specifying the effect of assignment to overlaid or based variables. For example, in the following program is the "correct" output 1 or 2?

```
OVERLAY x=y;
x := 1;
y := y+1;
PRINT x;
```

A rational language specification can be conceived to allow 1 as the only correct answer, or 2 as the only correct answer, or either as the correct answer; i.e., within certain constraints, the effect of the assignment is undefined. A SEMANOL specification can fairly readily mechanize the first alternative, will require a modeling of the entire storage environment to realize the second, and will probably fail to achieve the third in a way that does not impose a rigid definition not intended by the language designers. SEMANOL will undoubtedly produce a single,

[2] "An Axiomatic Definition of the Programming Language PASCAL," C.A.R. Hoare and N. Wirth, Acta Informatica 2, 335-355 (1973).

unambiguous answer which would be perfectly correct for a particular compiler but perfectly misleading for a mechanized language specification.

The argument for preventing overspecification should not be construed as a case for loose specification. It is simply that a SEMANOL specification, like any compiler implementation, ultimately provides some unambiguous semantic realization for each language construct. These implementation choices must not usurp the prerogatives of the language designers.

As stated earlier, good language design will minimize the undefined and surely minimize the impact of the undefined on program portability, intelligibility, and transparency. The "holes" remaining in particular language definition may give the compiler implementers extra degrees of freedom. This is an important factor in generating an efficient representation of the source language and it has no adverse effect on the users of the language. It seems unwise and costly to specify the semantic effects, for example, of optimization. Since the SEMANOL specification is by its very nature unambiguous, its adoption as a standard that would force each compiler to accept only the SEMANOL-defined language would establish an impractical goal.

6.2.3 READABILITY

As a language specification standard, it must be judged that a SEMANOL description fails in readability. For the language user, it is unsuitable as a reference document; for the compiler implementer, it is difficult to read. To learn the semantic effects of a particular language construct, it is necessary to review the extremely complex machine that describes it. This means following function calls through deep nestings and convolutions for even the simplest effect. Compared to a narrative description, it is noticeably deficient. An application programmer could not be expected to use a SEMANOL specification as a language reference document. An English-language description of a language can achieve the same objectives of completeness and unambiguity and result in a much more readable document.

For the compiler implementer, much of the same objection remains. Of course, he is more experienced in the jargon of languages, language definition, and compiler technology. Thus, the difficulty of the specification is not impenetrable, but a concise prose description would serve him better and at less expense. The way the semantics are necessarily realized in SEMANOL is largely foreign to compiler technology, thereby requiring a much more complex translation from the specification to the implementation. The SEMANOL specification serves to obscure that which the compiler writer needs to have most simply and clearly stated.

There is no intrinsic limitation to the clarity, completeness, or unambiguousness of an English-language description of any programming language. Furthermore, the effort to achieve such a specification cannot be expected to be more than that of writing a SEMANOL description (it is probably a good deal less); the result is considerably more useful for most readers.

6.2.4 VERIFICATION

Another objective of SEMANOL is to provide a machine-verifiable language description. However, the verification is not accomplished easily, nor are there any features of SEMANOL that facilitate the verification. The process is to execute test programs written in the language under measurement. Such a process is in no way different from testing of a compiler for the same language, except that the SEMANOL interpretation is much slower.

In particular, the degree of verification is a function of the thoroughness of the set of tests. If the tests lack comprehensiveness, then the SEMANOL specification is to that extent unverified. Furthermore, there is the requirement for a previously derived standard of success; i.e., do generated results match expected results? Again, the design of the test cases and their expected output is an activity independent of the specification writing. Relying exclusively on the SEMANOL description to define the results would not accommodate the possibility of error in the SEMANOL specification. In this situation, the quirks of a particular realization of a language specification - whether it is a compiler or a SEMANOL description - would wrongly become a default standard. If, in such a case, compiler results differed from SEMANOL results, which would be accepted? The SEMANOL specification is just as prone to implementation oddities and errors as is a compiler. Consequently, either the choice is arbitrary, or any specification must submit to yet a higher authority - a set of exhaustive test programs - for validation. Thus, SEMANOL does not appear to advance the state of compiler verification.

6.2.5 MACHINE-INDEPENDENCE

A useful language specification distinguishes between machine-dependent and machine-independent semantics. A well designed specification, furthermore, greatly minimizes the former, leaving undefined much that is machine-dependent. Clearly, a compiler is target-machine dependent to the extent that it generates machine instructions for some particular machine whose effects define the "undefined" semantics.

The concept of SEMANOL being more target-machine independent than a compiler is questionable; some 25 to 30 percent of the SEMANOL specification of J73 is given over to machine-dependent semantics. This is quite comparable to the DECSYSTEM-10 J73 compiler, for example. Does this mean that 25 to 30 percent of the language specification standard is to be rewritten for each new target machine? It is doubtful that this would improve upon a single English-language description, and there would be a heavy cost burden in preparing and verifying a SEMANOL description in addition to a compiler for each new target. The alternative of a single SEMANOL specification would incur implementation difficulties analogous to the JOVIAL/J-3 "signed-magnitude effect" problem, and would by its nature either invite deviations or impose excess costs on all development efforts.

6.2.6 MODIFYING THE SPECIFICATION

A language specification must be capable of easily accepting convenient and cost-effective modifications to accommodate language changes. A SEMANOL description is more difficult to modify than an English description. A somewhat serious limitation in SEMANOL is that the semantic definitions refer to constructs in the context-free syntax by their ordinal position (e.g., $\#SEG_n$) in the production. This can mean that even a simple change may necessitate the rewriting of each semantic function which refers to an altered syntactic production. Perhaps some future version of SEMANOL will lift this restriction by permitting the naming of elements of a production; however, this will impair the readability of Part 1, the most readable portion of the specification.

6.2.7 CURRENT STATE OF SEMANOL

Because of funding limitations, the testing of both the SEMANOL (73) program and J73 specification are incomplete. Furthermore, at this writing, the SEMANOL system is inoperative. The fact that twice during the development of SEMANOL, the system was disabled by modifications to MULTICS FORTRAN, leads to the conjecture that the implementation has relied too heavily upon MULTICS-dependent FORTRAN peculiarities and especially that FORTRAN I/O statements have been used too liberally throughout the program rather than concentrated in a single I/O module. These are, of course, implementation questions that are subject to improvement.

6.2.8 EFFICIENCY

Because SEMANOL employs the interpretive approach to program execution, testing is extremely slow. As a research tool, this is of relatively little concern. However, neither of the SEMANOL specifications thus far written have reached a satisfactory conclusion, and the extreme inefficiency of the system - aggravated in part by the inefficiency of MULTICS itself - must bear at least part of the responsibility for this. The proper verification of the J-3 description, for example, would demand exercising at least the same set of JCVS tests used to validate the JOCIT J-3 compiler. These tests are voluminous, and the amount of computer time required to execute them will render any testing activity quite costly. In addition, there is the problem of limited capacity of the SEMANOL system, which would require repackaging the JCVS tests into smaller modules, each with a considerable overhead in data definitions, thereby adding further to the required testing time.

6.2.9 DEFICIENCIES IN EXISTING SPECIFICATIONS

Two deficiencies in the existing SEMANOL specifications for JOVIAL cannot be overlooked. The first is the inability to handle COMPOOL, and the second is the failure to support independently packaged procedures. Perhaps both of these omissions will be corrected in future implementations, but for the present, these shortcomings obviate certain classes of testing, thereby leaving the specifications less complete than required.

6.2.10 EFFORT TO WRITE

The authors of SEMANOL have estimated that to prepare a J73 SEMANOL specification requires approximately 1.5 man-years. This in itself is not a prohibitive cost. However, since the specification must be modified (25 to 30 percent) to accommodate machine-specific semantics, this cost must be added to the effort to prepare each retargeted compiler. A conclusion that surfaced during the study of these costs is that this effort is not significantly less than that of writing a compiler for the same language that generates a simple machine-independent macro code that could be assembled and executed on the target machine. Such a small compiler can be chosen as a language standard to the same extent as the SEMANOL specification. Of course, it is less readable as a language standard than even the SEMANOL specification, but since neither is within acceptable limits of readability, such a comparison becomes irrelevant. The advantage of the small compiler approach is that at least a programming tool is produced. While not efficient, the tool is at least not interpretive and thus considerably faster than the SEMANOL specification. CSC does not propose the small compiler approach as an alternative to the SEMANOL approach; neither approach is relevant to providing either a standard readable definition or a compiler implementer's guide. Although the estimated effort to create a SEMANOL specification is not excessive, it has not been confirmed with a completed, comprehensive specification. Certainly more information is needed to judge the cost of the approach. Indeed, the preceding paragraphs have presented substantive objections that question the worth of the approach irrespective of cost.

6.2.11 SEMANOL AS A TOOL FOR LANGUAGE DEFINITION

SEMANOL seems eminently well suited as a tool to assist language designers during the developmental stage of language definition. For example, a SEMANOL specification can be developed alongside of the prose descriptions that evolve as language constructs are chosen and refined. The preparation of the SEMANOL specification undoubtedly will reveal semantic difficulties not evident in early discussions, and this will serve as useful feedback in the design effort. Of course, this does not imply that just because a particular semantic effect is hard to model in SEMANOL that this argues against the construct in question, but it is clear that simply imposing the discipline of mechanizing the abstract language ideas will assist in eliminating ambiguities and perhaps lead to useful simplification without diminishing language power. Furthermore, the mechanization process should result in a more complete prose description of the semantics which will serve both implementers and users alike. While the use of SEMANOL in this manner is outside of the notion of providing an LCF for some existing language, if some future LCF is undertaken for a language under concurrent definition, SEMANOL may be a useful tool.

6.3 CONCLUSIONS

SEMANOL is an interesting and thoughtfully conceived language tool. However, within the context of an LCF designed to support existing languages, it is largely peripheral to the central tool requirements. However, in the event that some new language is designed under the auspices of the LCF, then SEMANOL should be included as a language definition tool.

The deficiencies of existing language specifications must be corrected for the HOL or HOLs chosen for a pilot LCF implementation in order to provide a language reference manual and tutorial guide for applications programmers. Then, too, if an HOL is chosen for which no standard implementation exists that satisfies the LCF needs, the standard reference must be enhanced to serve the compiler implementers as well. These specification requirements are discussed further in Section 10.

SECTION 7 - RECOMMENDED LCF TOOLS

The tools to be incorporated into the LCF must meet the requirements of supporting (1) program development and (2) language and compiler development, modification, and maintenance. To meet these goals, the following tools are recommended for inclusion in the LCF:

- Compiler implementation tool
- Language specification
- Statistics collection
- Compiler validation
- Program validation
- Program development and implementation tools

These tools are discussed in the following paragraphs.

7.1 COMPILER IMPLEMENTATION TOOL

An HOL control facility must include a means for developing and maintaining a standard compiler for the language. This means that when new target or host computers become subscribers to the facility, a low-cost method must be provided to permit the rapid development of compiling facilities for the new machines. This requirement includes both rehosting and retargeting of the standard compiler. With certain modifications, the existing JOCIT system meets this requirement. The incorporation of JOCIT into the LCF is discussed in Section 9.

7.2 LANGUAGE SPECIFICATION

In order to support the HOL, the language must be defined in clear, complete, and unambiguous terms. The language specification must address two distinct readerships: (1) the application programmer, i.e., the primary user of the HOL, and (2) the compiler implementer and maintainer who is responsible for developing, enhancing, retargeting, rehosting, and maintaining the compiler in a reliable state of service for all subscribers to the facility. Two levels of specification are required for the users: a tutorial manual for newcomers to the language, and a reference manual that includes a complete semantic description of the language for line programmers. This second text serves as a proper requirements specification for the compiler implementer and maintainer as well. The development of the language specification is conceived entirely as a documentation effort; the SEMANOL system is not recommended for this purpose. The requirements for the language specification are discussed in Section 10.

7.3 STATISTICS COLLECTION

To assist in analyzing the use of language features and to provide objective data to help assess language deficiencies, language redundancies, and language difficulties, it is proposed to include in the LCF a means for the automated collection of both compile-time and execution-time program statistics. This proposal is based on the work begun for the JLMT (JOVIAL Language Measurement Tool) discussed in Section 3 and extended to provide functional specifications for its realization. The statistics collection tool is described in Section 11.

7.4 COMPILER VALIDATION

A successful LCF will include a means of validating that the standard compiler adheres properly to the language specification. To this end, Section 11 discusses the requirements for a validation system that will be used to verify the HOL standard compiler and also subsequent rehosted and retargeted versions of the compiler that are developed with the compiler implementation tool. For a J-3 or J73 LCF, the existing JCVS tests may be incorporated with enhancements aimed to correct the deficiencies observed in Section 5.

7.5 PROGRAM VALIDATION

The previous four tools are directed primarily to the goal of language and compiler standardization. To assist the development programmers in their tasks, the LCF will also include facilities for automated program validation. The objective here is to provide a standard, machine-independent, but HOL-oriented, tool that assists the programmer in systematically exercising all paths of his program and in providing him with symbolic data to display results which are a function of asserted values. Futhermore, statistical data will be derived from program executions which permit individual and composite performance analysis. This tool will be used by the programmer both at the source program symbolic level before compilation and after execution for postmortem performance measurement. Section 13 discusses the incorporation of JAVS into a J-3 LCF and provides a model for similar program validation tools for other HOLs.

7.6 PROGRAM DEVELOPMENT AND INTEGRATION TOOLS

The requirements for supplementary tools to aid both the applications programmers and the compiler implementer and maintainer are closely related to programming languages. Section 14 of this volume presents the requirements for source text editing, object module linking and loading, language translation, and debugging facilities. Traditionally, such tools have been supplied as part of the operating system and are developed with little regard to the support of HOLs and the kinds of tools suggested in this report. A pilot implementation of an LCF may go far in developing standards for these important supplementary tools and develop design goals that identify to what degree these may be implemented in a machine-independent manner.

SECTION 8 - LCF OVERVIEW

8.1 INTRODUCTION

Almost any contemporary operating system for a medium-scale to large-scale machine incorporates most of the previously described tools in some form. However, they generally fail to provide true HOL support, or if they do, a multiplicity of languages usually results in a compromise that favors support of only one at the expense of the others. For example, most systems do a reasonable job of supporting FORTRAN, but users of other HOLs on the system suffer the limitations of FORTRAN. These limitations include name qualifications only at the external module level, inability to refer to packed or part-word items, and the support for numeric statement labels only.

Furthermore, many of the tools have been built as an afterthought to certain unyielding aspects of the system architecture. It is not likely that any large, well-structured program exists that does not require the services of flexible linkage editing facilities. Yet, typical link editors are narrowly conceived, seldom addressed to program debugging interfaces, are usually sensitive only to a single HOL, often poorly documented (module formats, for example), and prone to error when exercising exotic but necessary facilities (such as overlays or common block placement).

A successful LCF will incorporate a variety of tools whose external specifications and interfaces submit to the principle of totally integrated design. Thus, the HOL compiler should not have to build symbolic debugging tables if the link editor cannot link them as well as the module text. Similarly, source program debugging of J-3 programs will permit the programmer to refer symbolically to local-scope variable names and packed items; furthermore, the displays will be by type, including status type. Although memory dumps are seldom used in a facility providing sophisticated interactive symbolic debugging, at least when it is required, such a dump should display module-relative addresses so that the programmer may easily find his program variables through reference to the compiler-generated environment listing. Thoughtfully integrated systems will also eliminate unnecessary duplication. It is hoped, for example, that the JAVS-generated execution summary files can be used for statistical analysis. The recommendations in this section assume such an integrated approach to implementing an LCF.

8.2 SELECTION OF AN LCF HOST SYSTEM

The tools studied for analysis for the LCF largely fail to meet the integrated design objectives; they simply were not developed with this principle in mind. Indeed, many were developed independently of the LCF concept. However, in considering a real implementation of an LCF, because of the availability of existing tools, a pilot implementation of a J-3 LCF appears to be very practical.

Considering the options for the system on which to build such an LCF, the lowest cost approach points to the HIS-6000 machines with GCOS. Fortunately, two of the tools, JOCIT and JAVS, operate on those machines. The GCOS design, however, does have a drawback in that it does not provide sufficient flexibility in its debugging and linkage-editing capabilities to best support the objectives of an LCF host selection incorporating integrated program development facilities. Section 14 of this volume describes the requirements for such services, and shows how GCOS responds to these requirements.

A reasonable alternative to GCOS is the MULTICS system. Because the MULTICS system operates side by side with the GCOS facility at RADC, it shares administrative and tactical advantages of an LCF based at Griffiss AFB. MULTICS offers the significant advantages of a totally integrated program development concept from the basic services of text editing (the weakest link) through file management services, link-editing, and debugging. The obvious drawback is that neither JAVS nor JOCIT are hosted in this environment and the cost of this rehosting must be measured against the cost of enhancing the GCOS services to bring them up to the requirements for the LCF.

Another disadvantage of MULTICS is its limited capacity (40 or fewer simultaneous users) and its slowness under even modest load (say, 20 users). Other problems exist in considering MULTICS, such as the MULTICS requirement that all software be written in PL/I. Inasmuch as JOCIT, for example, is written in SYMPL, rehosting is greatly complicated and the costs are significantly increased for rewriting JOCIT in PL/I. Furthermore, this conflicts with JOCIT rehosting design goals: it cannot be assumed that other hosts will support PL/I. However, the attractiveness of MULTICS is still manifest. Users of the system are exposed to - as is the case with few other systems - the best of integrated design concepts and a multitude of user-oriented services that support the diverse requirements of an LCF even beyond the concept of tool support; and MULTICS is, of course, widely available through the agency of the ARPA network.

8.3 FUNCTION OF TOOLS WITHIN THE LCF COMMUNITY

The interactions among the LCF community and the tools that support it are shown in Figure 8-1. The basis of any LCF is the language specification for the HOL being supported (or multiple specifications in the case of multiple HOLs). This specification is used by both the user community and the compiler developers, although as it has been suggested, it will exist in different forms for different levels of readership. Newcomers to the HOL will have access to a tutorial description, while production programmers will rely on the specification as a standard reference document for creating programs constructed with the HOL. Finally, the construction of the compiler implementation tool - JOCIT in Figure 8-1 - also is based on the same standard specification.

Of all the tools supported by the LCF, the compiler implementation tool is the most important. The success of the LCF is measured in terms of both the quality of the production compilers and the relative ease with which they may be

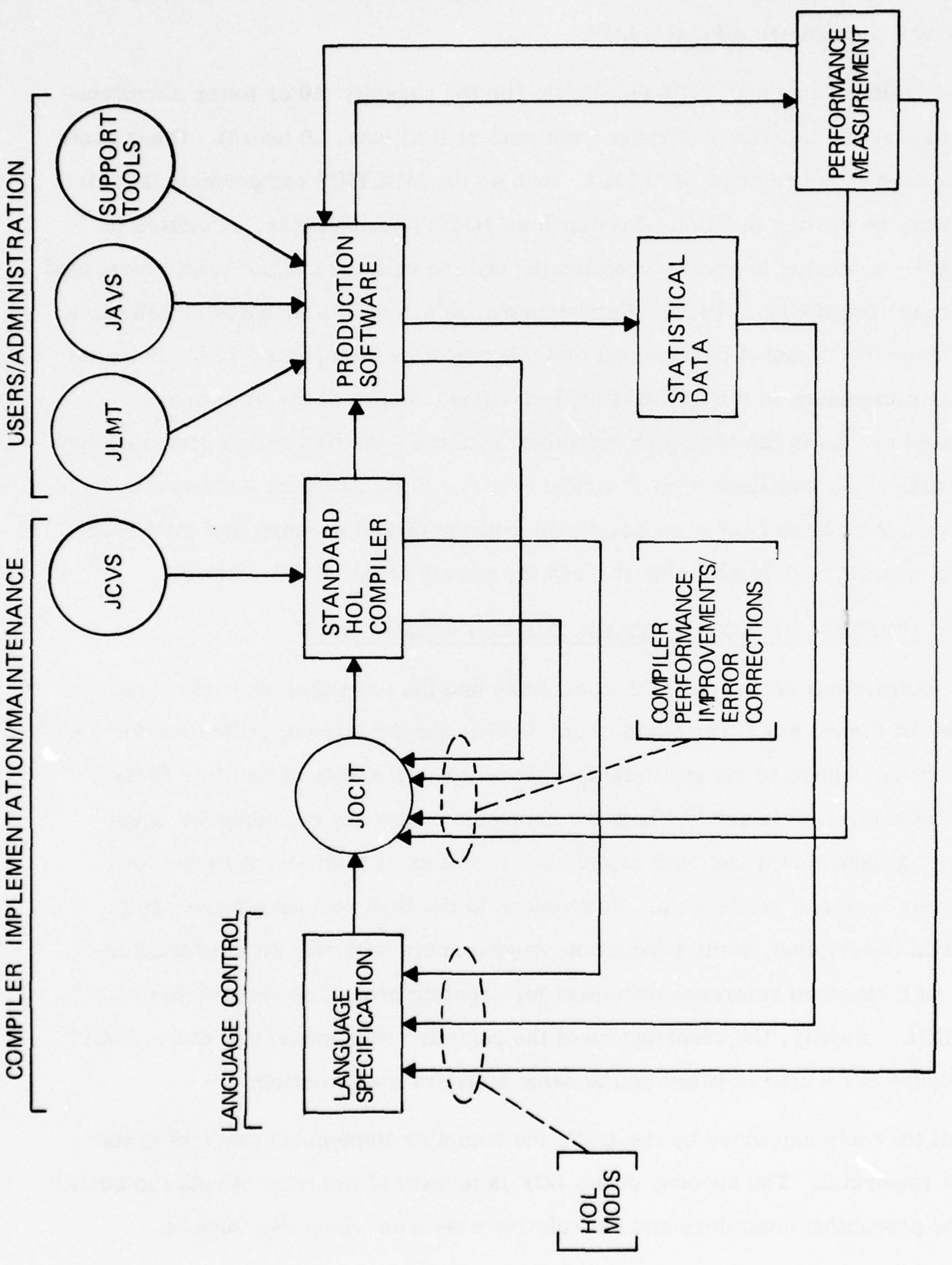


Figure 8-1. LCF Community Tool Interaction

constructed with the tool to meet new hosting and targeting requirements within the LCF community. Once the JOCIT component is completed, a standard HOL compiler is distributed. It may exist at a single site or at multiple sites; it may be hosted on a single machine type or on many different machines; and each copy of the standard compiler may generate code for any number of different target machines.

8.3.1 ERROR CHECKING

The correctness of the standard HOL compiler is verified using the compiler validation tool - JCVS in Figure 8-1 - and corrections to JOCIT, and thus to the standard HOL compiler, to complete a feedback cycle originating with failures detected with JCVS. This cycle terminates at a successful completion of the JCVS activity.

At any given site, then, the user programmers develop production software using the standard compiler. The major feedback within the LCF "system" originates at this point. Error reports are the most obvious form of feedback and may trigger the JOCIT/JCVS cycle once again until the errors are corrected. In a manual LCF, such feedback takes the form of written SPRs (software problem reports), which are routed from the source of complaint through the administrative center to the compiler maintenance team.

In an automated or partially automated LCF, such reporting may be greatly expedited by use of a central LCF timesharing facility (with or without the services of a network which only affects distribution; since this is a many-to-one transmission, the distribution function is not required). Reports may be displayed at prescheduled intervals at the LCF administrative center and routed almost immediately to the responsible maintenance agency.

The maintenance team need not be located at the LCF central site. They may employ periodic scanning of a "New SPR" file (e.g., on-line disk at the LCF) as a trigger for undertaking debugging. One aspect of this reporting-correcting cycle that requires the most thoughtful design is the access to data that caused the error in the first place. Failures in production programs are not easily reproduced at the maintenance site. Clearly, a network-connected LCF provides a necessary link between the source of the error and the activity that has been undertaken to correct it (or to determine that it is not an error in the software).

Other considerations appear then: How is access provided to the original files? Does the maintenance team require sanctioned accounts at each subscriber site? What happens when failure occurs in secure installations? These questions are not readily answerable in the abstract. It is most likely the case that procedures unique to each subscriber site will be required to solve this problem, and the solution may depart from the goal of full automation (e.g., tapes may have to be shipped from subscriber sites to the central LCF in order to permit the maintenance team to analyze the failure). Other solutions present themselves, but the generality of any of these maintenance interface specifications is limited. The full range of possible operating system architectures simply is not known.

One approach is to assign the maintenance team a highly sanctioned account at each subscriber site which gives them privileged access to user's files to reconstruct the reported failure. Read-only access may be sufficient for this purpose so that the files may be copied to the team's own account and thus preclude interference with production programmers. The advantage of such a scheme is that it relieves the users of further bother beyond that of reporting the error and ensuring that their files are not altered before the maintenance team can copy them.

Such an approach may not be applicable to subscriber installations requiring high security. It may not be possible to have access to the system at all, as is the case with certain military installations. It is beyond the scope of this study to deal with the complexities of security requirements of potential subscribers to the LCF, but swift response to user error reports is a clear objective of the facility. Perhaps secure installations could at least be provided with low-security access facilities. Then, disk packs or tapes may be carried from the secure area to a nearby companion low-security system connected to the LCF where the maintenance team could have dial-in access.

8.3.2 STATISTICS GATHERING

As is shown in Figure 8-1, the production facilities have direct access to two of the LCF tools: JLMT and JAVS. JLMT is the statistics-collection tool that operates both at user request and at administrative request. From the LCF point of view, the administrative function is the more important one. The HOL managers have the most vested interest in determining the effectiveness of the HOL and its compilers, and the raw data for determining this effectiveness is primarily the product of the JLMT. To this end, the gathering of language usage data must be controllable from the administrative organization of the LCF. The gathering of a meaningful sample of statistical data simply cannot be left to the chance invocation of the statistics-gathering tool by the users. Administrators need to be able to specify certain parameters of this activity, such as sampling frequency, programmer- or account-selection, data reduction cycle, and identification of the raw data repository. Ideally, a facility can be conceived in which the LCF administrator dials into a special account and sets these parameters on line or calls up certain report generators. This same administrator will have access to all subscriber sites for this purpose, and at some future stage of LCF implementation, the network connecting the subscribers will ideally permit electronic transferral of statistical data files from user sites to the LCF site for composite analysis.

Somewhere between user-selection of statistics gathering and automatic network inter-site data transfer lies a realistic goal for an immediately implementable LCF. However, even such a middle-level objective requires modification to the operating system discipline, and this may not be achievable at low cost. Furthermore, this exceeds the immediate capacities of agencies (such as RADC) to effect;

they simply cannot dictate changes to the operating systems to achieve the level of automated statistical collection believed to be the ideal requirement. Hardware manufacturers are rightfully protective of their operating systems, and they are not likely to make swift field changes to support the suggestions in this report.

The lowest level of statistics collection that excludes the willing participation of the user will be built into the standard HOL compiler. This method will append statistics to a system file to be consolidated periodically - asynchronously and selectively - and copied to secure storage. This scheme is fairly simple to implement and requires no tampering with the operating system in most cases. However, it is easily defeated by the user who has the access to the same system files as does the compiler. The user can corrupt the files any time, either knowingly or unwittingly. These alternatives are discussed further in terms of implementation design specifications in Section 11 of this volume.

The administrative function of the LCF will collect statistical data at periodic sampling intervals (a tunable parameter) and will create summary reports as described in Section 11. From these data, it may be determined what features of the HOL are underused, troublesome, misapplied, etc. Picking the right data to collect is the key to the entire collection issue. Administrators may present the data to the language change control committee who will have a factual basis for making certain language changes. It would be interesting, for example, to discover to what degree parallel FOR statements are actually employed in real JOVIAL/J-3 programs. They are not easy to implement, potentially troublesome, and largely redundant.

The proper application of statistical analysis will be most effective if tightly controlled at two levels of the LCF: the administrative level, and the language control level. The LCF administrators will control the raw data collection, subsequent data reduction, final report generation, frequency of report distribution, and report form and content. The language change control committee will meet periodically, perhaps in response to some pressing need identified by the LCF administrators from their study of the statistical reports (as well as from hearing complaints of users). The change control process will, as a result of LCF implementation, use the usage data collected from subscriber organizations within the LCF community to temper natural biases and the inclination to accept a loudly advocated change. There is no better way, for example, to stifle an objection to the deletion of type "dual" from the J-3 language than to present a year's statistical data showing that it was used, for example, in less than .0001 percent of all compiled lines.

Statistical data also will be used in assessing compiler performance. Execution statistics can greatly influence the choice of areas in which to concentrate optimization, the selection of table-packing algorithms, the degree to which static storage sharing is attempted, or the choice of fixed or stack allocation for local-scope non-OWN variables. Statistics from different user environments may show that some of the above choices should be incorporated as user-selectable alternatives in the standard HOL compiler. Some future version of the LCF may provide for direct feedback of certain execution statistics to the optimizing phase of the compiler so that the application of optimizing algorithms may be automatically and instantaneously responsive to changes in the user program profile.

Because HOLs are procedure-oriented, a typical, well structured programming system employing the HOL will actually describe yet a higher-order language in which higher functions and operations are described as user-written procedures. The higher-level function of such procedures is not directly inferrable from an automated examination of the code (consider, for example, list-processing subroutines in JOVIAL). Therefore, statistics collection based on language forms must be supplemented by additional information in order for the data to show the ways in which the HOL (or HOLs) should grow to meet real processing needs. For example, should variable-length strings, or list-processing operations be incorporated as language primitives?

8.3.3 TESTING

JAVS is a companion tool to the statistics collector. The principal difference is that JAVS is invoked exclusively by the user to aid him in validating his programming system. The implementation of JAVS is largely exterior to the operating system and thus gives the LCF developer greater freedom than does the statistics collector. JAVS works almost exclusively with user files, operates at the user's request, and creates output for the user's consumption and analysis. Since JAVS aids the user in verifying the program against its design specifications, it is a primary aid to LCF administration in assessing the performance of the HOL.

Whereas error statistics can reveal troublesome HOL features, total program development difficulties typically result from several factors - poor programming techniques, incomplete and incorrect design specifications, shortage of necessary programming skills, and inapplicability of the HOL to the programming task. However, the importance of JAVS to the user is significant in an orderly development of a complex system. It points the way to a logical and comprehensive testing strategy that aims to test all paths of the system in a systematic, measured way.

There is one aspect of JAVS implementation that dovetails usefully with the statistics-collection function: the recording of run-time data. The same JAVS-generated data base may be used by the statistics collector to determine run profiles, which may be displayed against source listings to determine heavily used program constructs, to provide correlation between program features and execution failures (program aborts) or errors (failures of the program to produce expected results). Ultimately, program managers may infer from their development experience just what aspects of the HOL merit enhancement, clarification, correction, or optimization. Thus, a JAVS-like tool assists indirectly in the LCF function of maintaining a serviceable and up-to-date HOL that addresses the real needs of the community.

8.3.4 OTHER SUPPORT FUNCTIONS

The final set of support tools are so classified because they are necessary adjuncts to the LCF mainstay tools, but do not directly serve the principal LCF requirements. They are proposed as integrated members of the LCF to strengthen the principal tools: the compiler building tool, the compiler itself, the statistics collector, and the program validator. They provide no direct feedback to the language specification function nor do they influence the growth and development of the HOL. However, because of the integrated approach, they may dictate certain functional amendments to the basic tools (e.g., additional compiler output for debugging).

While most of the proposed support tools are standard and rather obvious, the inclusion of the language translator tool is not. A single HOL LCF has its greatest appeal to potential subscribers whose programming task is planned in that HOL. For those installations who intend to program in a different HOL, the LCF may be irrelevant. However, there is yet a third class of potential subscriber who operates a well-developed program complex and is at the stage of transition into a major "next generation." He has a large investment in programs that he cannot afford to simply discard, but he sees the advantages for his program in the use of the HOL. How does he justify a language switch? The LCF can service his needs by providing a number of language translators. Thus, for example, if an installation is considering discarding JOVIAL/J-3 in favor of J73/I, his decision is made a good deal easier if the LCF offers a source language translator from J-3 to J73/I.

HOL translation is clearly not an easy task, but its difficulty should not necessarily defeat attempts to provide them. Some languages are relatively straightforward in their translatability to other languages (e.g., SYMPL/J73, J-3/J73), and even the difficult ones can be done with perhaps only a small sacrifice in efficiency. COBOL, for example, does not readily translate to J73 on a one-to-one basis, but thinking of the I/O statements as procedure calls opens the possibility of successful translation.

Other features, such as decimal-type variables, offer significant efficiency problems in translating from PL/I or COBOL to, say, FORTRAN or JOVIAL. The problem there is really not one of translation - decimal values may also be manipulated by subroutine, which they probably are by the PL/I or COBOL compilers anyway - but rather one of applicability of the LCF's HOL to the potential subscriber's programming problem. If his programs really require extensive use of decimal arithmetic, then perhaps J73, for example, is simply not a cost-effective choice in the first place. Nonetheless, machine translation has been effective in a variety of special situations.

The highest probability of success occurs in translation between two languages on the same machine. A major pitfall in translation is the accommodation of hidden machine-dependencies coded in the HOL, and these are almost impossible to recognize and handle effectively at the translation level. A reasonable goal of language translation is to achieve something less than 100 percent automatic translation. An LCF will be the more inviting to potential subscribers if translators exist and their effectiveness can be demonstrated in the form of benchmarks.

SECTION 9 - COMPILER IMPLEMENTATION TOOL

JOCIT provides a basis for a generalized compiler implementation tool. To realize JOCIT's maximum potential within the LCF, certain enhancements are proposed to facilitate the principal functions of convenient rehosting, retargeting, and meeting the interface requirements of the support tools. The following subsections discuss these proposed enhancements.

The presentation assumes modifications to the existing JOCIT to support a JOVIAL/J-3 LCF. The majority of the recommended changes are, however, entirely compatible with any other HOL chosen for the LCF; indeed, GENESIS was chosen for front-end development to support both convenient rehosting and multiple languages. The retargeting, rehosting, support tool interfaces, and improved optimization discussions are, with one exception, entirely independent of the choice of HOL. The one exception relates to the use of SYMPL as the compiler implementation language.

SYMPL was chosen as the implementation language to facilitate rehosting. A compact, reliable compiler existed which enabled quick bootstrapping, and the language itself is far superior for compiler writing than JOVIAL/J-3. However, in considering the possibility of a J73 LCF - the pressure for J73 standardization is clearly mounting - experience in developing the DECSYSTEM-10 J73/I compiler has proved that this language is at least as well suited to compiler implementation as is SYMPL. Thus, a J73 compiler is more usefully implemented in its own language, and the principal effect is to simplify the development logistics considerably. Thus, the two subsections entitled "Retargeting SYMPL" and "Optimizing SYMPL" (subsections 9.2.2 and 9.6, respectively) are relevant only to the implementation of a J-3 (or any other language not suitable for JOCIT implementation) LCF.

Subsection 9.5, Automated Code Generation, has been included in the tool concept. However, the technology proposed is considerably more speculative than that relating to the other enhancements, and should, if adopted, be undertaken as a parallel R&D effort. There is much to be gained in exploring this largely new topic. However, even a partially successful venture would pay significant dividends in meeting the main objectives of the LCF and particularly in supporting the real needs of the user community.

Integrating Requirements For Generalized HOL support (subsection 9.7), addresses the required enhancements to support languages other than JOVIAL/J-3. While the choice of a J-3 LCF reduces implementation costs by adopting the existing tools (JOCIT, JAVS), the choice of J73 or some other HOL requires new development effort. The goal of integrated tool development, then, is best supported by incorporating many of the separate functions identified for the statistics collection and program validation tool into the framework of the compiler implementation tool. This integration is discussed, and the principal new components are identified for integration into the JOCIT program.

9.1 ENHANCEMENTS FOR RETARGETING

To make the LCF more inviting for a potential subscriber whose machine is not supported by the HOL compiler, a means must be provided for quick retargeting of the compiler. Using an HOL often has been rejected because no compiler was available for the HOL. Quick retargeting effectively eliminates this objection and can even give the potential subscriber the facility to run benchmarks to aid decision-making. The objective of quick retargeting is not consonant with generating efficient code, and the retargeting enhancements are suggested with this limitation in mind. However, CSC's proposed approach rejects such schemes as interpreted object code whose performance is so deficient as to neutralize the attractiveness which the quick retargeting facility presents. The objective of retargeting is to make available quickly a compiling capability for any new target machine for which an assembler and loader are available. While the JOCIT compiler is basically well suited to meet these retargeting goals, certain enhancements are proposed to render retargeting less costly than at present and to reduce the required schedule as well. Applying the following modifications to JOCIT should yield a retargeting effort within the range of three to six man-months.

9.1.1 TARGET MACHINE PARAMETERIZATION

Several JOCIT modules make implicit assumptions about characteristics of the target machine architecture. These must be identified and replaced by references to standard compiler-wide target machine parameters. Typical of the necessary parameters are:

- Word size
- Byte size
- Bytes per word
- Medium-packing access field descriptions
- Addressing units per word
- Standard alphanumeric collating sequence

9.1.2 QUICK CODE GENERATOR

The present JOCIT code generator and editor phases must be supplemented by new quick code generator (QCG) phases. As conceived, there is a COMPOOL-like file that incorporates the description of each supported target machine. The QCG embodies both code generation and editing functions. The QCG may treat IL sequences generated by the front end as macros and generate "canned" code sequences for them. The editing function is somewhat simplified in that source code output, rather than relocatable binary, is generated. Such a QCG scheme, while not exactly automatic, is of low effort once familiarity with the rather rich JOCIT IL is achieved. To minimize the effort for each QCG, it is suggested that the quick-compiling mode of the JOCIT compiler imply no optimization; i.e., the optimizer phases are not invoked in quick-compile mode. This eliminates the complex VALD/VALU/SPOIL interface and the attendant requirements for register memory maintenance.

9.1.3 EXECUTIVE MODIFICATIONS

Required executive modifications include the recognition of the "quick compiler" option, selection of QCG instead of COGEN and EDITOR, and provision to generate the assembly source output of the QCG phase.

9.1.4 JOVIAL LIBRARY

To avoid rewriting the J-3 library for each new retargeting, a J-3 LCF should contain a version (perhaps the only version) of the library that is itself coded in J-3. It is impossible, of course, to write the library entirely in JOVIAL, but as much as possible should be so written. Thus, the library may be compiled in the quick-compile mode of JOCIT, which will make the library available at low cost. There will be some loss of efficiency by taking this approach, but that is consistent with the quick retargeting goals. Ultimately, the library can be rewritten in target-machine assembly language to achieve maximum efficiency.

9.1.5 DIRECT CODE

The direct code processor subphase of the analysis phase must be replaced by a direct code processor that accepts target machine assembler source. This can be achieved by a new processor for each such target or by placing certain constraints on the reference to JOVIAL variables within direct code and providing a canonical direct code processor. The latter approach is more costly to develop but reduces subsequent retargeting costs somewhat. The present intermediate file direct code interface is sufficiently general to support either approach.

9.1.6 TESTING

Testing of any new retargeted compiler should consist of exercising the JCVS (or its equivalent) series on the new target machine. These standard tests are parameterized for target machine data characteristics and, as such, are readily portable to the new environment.

9.2 ENHANCEMENTS FOR REHOSTING

The principal requirement for the compiler building tool within the LCF community is its own rehostability, a much more complex task than retargeting. The compiler itself, consisting of over 50,000 lines of source HOL must be compiled, moved, integrated, and tested. The main element of rehostability is that the compiler is itself written in an HOL (SYMPL). To a lesser extent, host machine independence is achieved through minimum assumptions within the compiler on host machine characteristics. These assumptions must be considered in order to give the tool maximum rehosting potential. An irreducible number of steps are required to rehost a JOCIT compiler:

- The JOCIT executive must be installed on the new host
- The SYMPL compiler component of JOCIT must be retargeted to the new host
- GENESIS must be modified, as required, to produce tables consistent with the host machine data formats
- All JOCIT components must be compiled with the retargeted SYMPL compiler
- The compiled components must be integrated on the new host with the new executive
- The compiler must be validated, which may involve recycling of some of the above steps (e.g., serious errors in the SYMPL compiler output may require recompilation of the entire JOCIT system)

To meet the objectives of this minimum set of steps, the enhancements to JOCIT described in the following paragraphs are required.

9.2.1 JOCIT EXECUTIVE

The executive is written almost entirely in assembly language for compiler efficiency. This requires rewriting for each new host, traditionally an effort of at least six man-months. This effort may be reduced by rewriting some executive components in an HOL, either JOVIAL or SYMPL. For maximum HOL-independence, it is suggested that SYMPL be used, since retargeting SYMPL is required in any case, while retargeting JOVIAL is not necessarily implied by rehosting JOCIT. (However, most rehosting efforts will probably include retargeting of JOVIAL for the new host as well.)

9.2.2 RETARGETING SYMPL

The SYMPL compiler component of the JOCIT system is divided into two phases: the syntax analyzer and the code generator. Retargeting requires replacing the code generator (and its included editor). It is proposed that such a code generator be built along the same architectural lines as the present generator. This includes some moderately complex local optimization that extends the retargeting effort, but the payoff is improved JOCIT compiler performance. Where compiler performance is less important than the need for low-cost rehosting, a generator for SYMPL that follows the design goals identified above for the JOVIAL QCG will suffice.

Rehosting the SYMPL compiler is not necessarily required for retargeting it. However, where the old SYMPL host is significantly removed from the new or where the mechanics of transmitting the output of the SYMPL compiler to the new target are sufficiently tedious, rehosting SYMPL may be more expeditious. Since the SYMPL compiler is itself written in SYMPL, it furthermore provides an excellent method of validating the retargeting. Additional effort may be required to effect full rehosting, such as data base adaptation, for example. Typically, rehosting JOCIT will be greatly facilitated by rehosting SYMPL, and the additional effort is relatively low.

9.2.3 ADAPTING THE JOCIT DATA BASE

Adapting the JOCIT data base to a new host whose word size differs from that of any other JOCIT host (there is only one at present although JOCIT is only a single generation evolution from compilers which run on three other hosts) requires manually adjusting many of the structure declarations to use space efficiently.

Subsequent rehosting could be simplified by including in SYMPL a provision for SYMPL compiler-optimized data allocation. This would entail a change to the present SYMPL language in which the programmer would identify logical structures and the individual items to be shared with other logical structures. The SYMPL compiler would then proceed to produce an optimum packing for the specification. Such a procedure is extremely complex if a straightforward combinatorial approach is to be avoided. (For a structure such as the JOCIT symbol table with more than 100 items and 30 logical substructures, the computation time exceeds manageable limits.) Such a structural optimization scheme is not seen in contemporary HOLs, but it is relevant to the portability question. CSC recommends its inclusion in the SYMPL language.

9.2.4 GENESIS MODIFICATIONS

The output of GENESIS is in the form of SYMPL source language array, item, status list, and switch declarations. Only the first of these is affected by rehosting. The sensitivity in GENESIS to the host machine is limited to word size. GENESIS currently includes the provision for several common word sizes, but for some new host not accommodated by this set, modifications will have to be made to support it. This is a somewhat larger task than simple parameter adjustment, for GENESIS goes to great length to pack the coded syntax tables optimally.

9.2.5 HOST MACHINE CONSTANT FORMATS

A few components of the JOCIT compiler are sensitive to the host machine word and arithmetic architecture. This choice is made to favor compiler efficiency over portability. For example, the optimizer performs constant arithmetic and other similar optimizations using host machine constant formats and host machine arithmetic directly. For improved portability, another scheme employing canonical form constants (including character constants) is required. Such a scheme was employed in the first-generation GENESIS-based ALGOL compilers which compiled on three hosts for six different targets. The resultant increase in portability will, however, be achieved with some slightly reduced performance in the front-end and optimizer phases of the compiler.

9.3 SUPPORT TOOLS INTERFACE

If the proposed support debugging tool is implemented for the LCF, the JOCIT compiler will be required to generate a composite debugging dictionary (CDD). The form of the CDD depends upon the system chosen for LCF implementation. For a MULTICS LCF, the MULTICS debugging interface would be used. If GCOS is chosen, the shortcomings of the GCOS debugger are best overcome by development of the debugging tool suggested in Section 14, and the CDD will conform to the detailed specifications developed for that debugging tool.

The required changes to the JOCIT compiler to generate the CDD are almost entirely embodied in the EDITOR phase. (Note that the CDD would not be generated in quick-compile mode if assembly language output is generated.) For known debugging systems (e.g., DECSYSTEM-10/DDT, INFONET-CSTS/PCF), the CDD is largely a reformatting of the compiler internal symbol table which can be performed by the EDITOR phase as a postprocessing function. The statement descriptor table can also be generated by the EDITOR as it reads the code file. The MULTICS debugging interface presents some unique requirements that will involve the cooperation of earlier phases of the compiler: namely, the facility to reproduce the source statement during interactive debugging will require the front end to pass to the EDITOR a file of source code indexed by statement number.

Source text editing may require some additional changes to the compiler for most convenient user interface. The ability to deal with source file keys (the compiler should ignore them but retain them for the listing) is one such change, and it might be useful to consider reconstructing the set-used listing to refer to such keys rather than to compiler-generated line or statement numbers.

9.4 IMPROVED OPTIMIZATION

Since the JOCIT design incorporates a machine-independent optimizer phase, enhancements to it do not interfere with the retargeting and rehosting goals. Once implemented, machine-independent optimizations are automatically available for the full range of supported hosts and targets (except in quick-compile mode discussed in Section 9). The following optimization enhancements are discussed in detail in Appendix B:

- Code straightening
- Dead variable analysis
- Extend loop optimizations
- Parallel path optimizations
- Use of COMPOOL by the optimizer
- Miscellaneous optimization enhancements

9.5 AUTOMATED CODE GENERATION

The present JOCIT compiler employs a code generation scheme that is totally modular (i.e., it can be readily replaced by a code generator for a new target machine) but it is tailored almost exclusively to the particular target machine in order to permit maximum special-case optimization. Some components of the code generator are, in effect, target-machine independent (triad table building, VALU usage/register memory management, for example).

While the quick code generation scheme permits rapid retargeting, it does not include any provision for either global or local optimization. For production use of the retargeted compiler, the resulting code efficiency may be insufficient to meet the target system requirements for either program size or execution speed. This deficiency is addressed by constructing a tailored, optimizing generator for the new target (one that responds to the global optimization IL transformations and also includes local, machine-specific optimizations). This effort may be substantially reduced by employing an automated code generator. This may be achieved through the following:

- Isolating machine-independent code generator functions
- Parameterizing certain machine-dependent code generator functions (such as register selection)
- Describing code file output formats in a stylized way to be processed by GENESIS
- Describing certain standard code generation functions (such as subroutine linkage conventions) as IL macros
- Describing the target machine as a set of state transition equations in terms of JOCIT IL operations

The last-mentioned target-machine description represents the heart of the automated code generation scheme. The equations will be used by a totally generalized code generator driver to derive the optimum sequence for computer expression (formula) code. The description is entirely straightforward and easily written from the hardware operations manual for the target machine. It will be processed by the GENESIS system, which will generate a set of tree-structured tables.

The code generator driver - the analogue of the front-end ANZR program - is written once for all target machines. The IL, as output by the front-end and transformed optionally by the optimizer, is input to the driver which operates on the IL string and the tabular description of the target machine to produce the code file output. Certain IL sequences will provoke somewhat straightforward IL macro expansion; the macros are also written for GENESIS processing, and the resultant tabular output will be available to the driver program. The driver will process the IL string much as the ANZR program processes the construct string. Certain recognitions will be made which will result in unconditional replacements and code file output until the IL string is exhausted.

AD-A035 915   COMPUTER SCIENCES CORP   HUNTSVILLE ALA   F/G 9/2
LANGUAGE CONTROL FACILITY (LCF) STUDY. EVALUATION OF THE SOFTWA--ETC(U)
DEC 76   W ANDERSON, T L DUNBAR, P L BELFORD   F30602-76-C-0024
UNCLASSIFIED   CSC-4549-110   RADC-TR-76-386-VOL-2   NL

2 OF 2
AD-A
035 915

END
DATE
FILMED
3-25-77
NTIS

A companion to this scheme will be the tabular description of the assembly language listing to be generated by the editor phase (much as was done in the original GENESIS ALGOL compilers). The interface between the automatic code generator and the editor - the Code File (CF) - will also be constructed by GENESIS; i.e., the CF array description, and the principal status lists to define the operations codes will be output as SYMPL source language. It will also be useful to investigate methods for describing relocatable object file formats in a form processable by GENESIS. However, it should be noted that in the past similar investigations have determined that manual coding of the object file generator is generally easier and much more efficient than describing the interface.

The implementation of the proposed automatic code generation scheme represents a significant advance in the state of the code generation art. For those installations considering the LCF to support their system development requirements, the combination of quick retargeting and automatic code generation for efficient production compilation is extremely attractive. The automatic code generation approach is proposed to reduce significantly the cost of subsequent generator development, but it cannot be expected to reduce it to the level of the quick code generator. There is also the development cost of the process, which must be amortized over the expected range of applicability.

9.6 OPTIMIZING SYMPL

The choice of the straightforward architecture of the SYMPL compiler component of JOCIT was made to facilitate rehosting. The SYMPL compiler attempts only a modest set of local optimizations in order to permit the fastest possible bootstrapping. For example, full bootstrapping of the J73/S compiler from the 1108 to the DECSYSTEM-10 was achieved in approximately 3.5 man-months. The bootstrapping was demonstrated by successful self-compilation of the compiler. Success was measured in terms of a bit-for-bit comparison of the object code of the entire compiler in both the nth and (n+1)th versions - a single bit difference was treated as a failure. The cost of this approach is somewhat impaired compiler efficiency. Both the size of the JOCIT compiler and resulting speed of execution of the compiler are subject to improvement. CSC therefore recommends that the SYMPL compiler be modified to permit the output of the SYMPL front end to be processed by the optimizer and code generator phases of the JOVIAL compiler. The recommendation includes the constraint that the quick-compiling facility not be impaired. The modifications to SYMPL would include:

- Adding a translator interphase to transform the SYMPL compiler IL to JOCIT IL and to build a JOCIT symbol table from the SYMPL symbol table
- Enhancing the JOCIT symbol table and IL where necessary to accommodate SYMPL language forms that do not exist in JOVIAL (e.g., based variables, scalar name parameters, pointer formulas, COMMON blocks, procedure parameters)

By way of comparison, the DECSYSTEM-10 J73/I compiler was increased in speed and reduced in size by approximately 20 percent when self-compiled with the optimizing DECSYSTEM-10 compiler versus the J73/S bootstrap compiler. This appears to be a significant enough improvement to merit serious consideration. Furthermore, the capabilities of the JOCIT global optimizer - especially when including the proposed enhancements - are considerably greater than those of the DECSYSTEM-10 regional optimizer. On the other hand, the current JOCIT SYMPL compiler - the parent architecture for the J73/S compiler - employs considerably more local optimization, so that perhaps less than a 20 percent improvement is realizable.

This section does not recommend elimination of the present SYMPL compiler, but instead, recommends supplementing it to provide the choice of the current level of generated code or a more optimized level.

9.7 INTEGRATING REQUIREMENTS FOR GENERALIZED HOL SUPPORT

The preceding sections have dealt with incorporating JOCIT into an LCF that is assumed to support the JOVIAL/J-3 language. While the majority of the JOCIT program is designed with language independence in mind, still there is a J-3 bias. Certain aspects of the symbol table structure and IL design and the choice of SYMPL as the implementation language reflect this bias. Furthermore, many of the functions defined for the statistics gathering tool and the program validation tool may be more profitably integrated into the HOL compiler. These considerations are addressed in the following paragraphs.

9.7.1 ACCOMMODATING MULTIPLE HOLS IN THE JOCIT DESIGN

Much of the JOCIT model as presently implemented is language independent. The optimizer, code generator, and editor phases are essentially free of language dependencies. Even the syntax analysis phases, while clearly JOVIAL-oriented, are nonetheless constructed using the GENESIS approach which is formally language independent. Of course, if a new language is to be supported, new syntax tables (GENESIS-constructed) must be created and new pragmatic functions must be written to support them, but the analysis program itself (ANZR) and all supporting I/O and symbol table services are entirely independent of the compiled language. In fact, the JOCIT design meets multiple HOL requirements to a significant degree, and the same compiler model could be used to compile any number of different languages. The compiler executive would select the appropriate syntax tables and pragmatic functions according to an HOL selection option on the compiler command line.

Certain aspects of the symbol table design reflect J-3 orientation. Mostly, this orientation is expressed in terms of special classes of symbol table entries that express unique J-3 forms (such as arrays, for example). Similarly, the IL includes operations (such as switch call) which are derived from J-3 requirements. These JOVIAL-specific features are largely superfluous, however, and could be expressed by more general forms than already exist, or they could be added as required to support other HOLs.

Many of these language-specific features were included to permit tailored optimization of JOVIAL, but the proposed optimizer enhancements render such language-specific forms (such as FOR-loop-specific IL) redundant. The generalized forms will be processed and optimized thus simplifying the front-end and all the interfaces. However, the accommodation of languages that differ markedly from JOVIAL will doubtless require enhancements to the symbol table and IL design, the requirements for which are not identifiable in the abstract. Of the known features of richer languages, such as PL/I, enhancements would be required to support recursion, name qualification, and dynamic storage allocation.

The use of SYMPL as the implementation language for JOCIT was chosen because of its clear superiority over J-3 as a compiler writing tool. However, for other HOLs such as PL/I or J73, this superiority is no longer evident. The complexity of requiring the support of two languages within the compiler implementation tool is no longer paid for when the principal HOL - or any one of, say, a family of HOLs supported by the same tool - is suitable for compiler writing. CSC recommends that a J73 (or similar HOL) LCF require that the implementation tool be written in its own language. Aside from the simpler logistics, the use of a single language implies self-compilation which, as pointed out earlier, comprises a thorough test of the compiler even before submitting the compiler for validation. However, a multiple-HOL LCF might choose to support all HOL software using a single HOL.

The choice of a single-language compiler implementation tool does not exclude, however, the employment of a bootstrap compiler for initial implementation. On the contrary, CSC recommends the use of the SYMPL compiler architecture as a base for a bootstrapping tool for initial development. If J73 is chosen as the LCF HOL, such a compiler already exists (the J73/S compiler operating on and generating relocatable object modules for both the 1108/CSTS and the

DECSYSTEM-10/timesharing systems). Since this architecture has been used successfully for production use of J-3 (subset), SYMPL, and J73 (level I subset), its flexibility has been demonstrated to the degree that these three languages differ. Its adaptability to subset PL/I for example (eliminating ON conditions, I/O, decimal arithmetic, and automatic allocation) is expected to be high because the resulting subset becomes more congruent with the predecessor languages (and the deleted features in no way compromise its suitability for compiler implementation, especially at the bootstrapping level).

9.7.2 STATISTICS COLLECTION

The static statistics collection function (addressed in Section 11) is most suitably implemented within the compiler framework. Static statistics may be collected by the analysis phase for language data and by the optimizer, code generator, and editor for object program data. The results may be optionally displayed at compilation end and/or collected for insertion as part of the compiler output, either in the object file, CDD, or special statistics data file, depending upon the choice of operating system, language, and implementation method.

For dynamic statistics collection and performance measurement (for program validation), the demands upon the compiler will depend upon the measurement scheme employed. Of the two approaches discussed in Section 11, the first requires that the code generator compute statement execution timing and insert the results into the CDD from which it may be extracted later for postmortem program profile analysis. If this approach is chosen, there will be some change to the code generator/editor interface to accommodate the timing information.

9.7.3 REFORMATTED SOURCE TEXT

The JAVS program presently provides source language reformatting for the J-3 language. The JOCIT compiler does not provide this service. For a non-JOVIAL/J-3 LCF, it is more economical to combine the source reformatting facility with the compiler syntax analysis phases. In order to integrate this facility with other tool requirements discussed in Section 14, the compiler will also include the facility to generate a reformatted source language file which is indexable by statement number. This file may be an appendix to the CDD, or it may be a totally separate source file. The choice is dependent upon other considerations, such as the file management characteristics of the operating system.

Consistent with the notion of integrated design, the option of reformatting should be related to other requirements. If the debugging package, for instance, is to support source statement reconstruction, then the reformatted output must accompany the generation of the CDD. The retrieval of source statements by statement number demands the reformatting approach described above. Furthermore, if statistics collection is to be controlled from a level higher than the programmer level and if the statistics collection function is supported by the CDD, then its generation (and thus the reformatted source generation) may not be optional at all.

9.7.4 PROGRAM VALIDATION

Two provisions of the program validation function as described in Section 4 should be directly incorporated in the HOL compiler. The first function is the flow analysis required to identify the DD-paths. Since the first pass of the optimizer phase performs flow analysis in any case, the construction of the flow tables describing the DD-paths falls naturally into the optimizer domain. This requires extension to the flow analysis process to accommodate the JAVS postmortem analysis requirements (i.e., to generate the formatted tables). If the code straightener option is selected as part of the JOCIT enhancements effort, then this function may be embedded in the straightener; otherwise, the present flow analyzer must be modified to provide the DD-path identification.

The second function, value verification - the ASSERT, EXPECT, and TRACE directives - should also be incorporated in the compiler by means of extensions to whatever HOL is chosen for the LCF. If such language extensions are not possible, then other methods may be found to effect the same results (for example, the use of parameterized DEFINE statements in J73).

Most of the preprocessing functions of JAVS, then, may be absorbed by the compiler, and this eliminates the redundant requirement for source scanning and passing. The remainder of the JAVS function - the report preparation - should be retained as a separate tool as this neither duplicates compilation functions nor falls under proper compiler responsibility.

SECTION 10 - LANGUAGE SPECIFICATION

The language specification for the HOL does not fall into the category of software tools. However, the need for a comprehensive and complete specification is clear. It is proposed that three different forms of language specification materials be developed:

- Standard reference document which includes all syntactic forms and the details of the semantics including the identification of the machine-dependent semantics which will fall into the categories of "implementation-defined" or "undefined-by-this-description"
- Standard tutorial text for newcomers to the language
- Standard teaching guide for instruction in the HOL

10.1 STANDARD HOL REFERENCE

There are many existing language specifications that are useful models from which to derive a complete reference manual for the HOL (or HOLs) supported by the LCF. The most complete among these are the PL/I "Vienna" description and the ALGOL 68 Report. Both of these descriptions suffer from much of the same criticism leveled at SEMANOL, namely, that their very completeness - and in the case of ALGOL 68, its conciseness - render them largely unreadable. However, the approach taken in the ALGOL 68 Report holds a good deal of promise. For example, the two level syntax is unnecessarily complex and opaque and could be replaced by a simpler BNF-type description supplemented by clearer expository prose to describe the semantics.

The problem with most prose semantic descriptions is the failure to choose a consistent model, or "machine," in which to express the effects. This shortcoming is relieved by the method chosen in the PL/I and SEMANOL approaches, but while the modeling is consistent, the resulting description is simply too dense and intricate to be useful. The ALGOL 68 "machine" is somewhat more informal and therefore understandable. The overly concise language, however, makes the report difficult to read. (This assessment refers to the style, not the form.)

Perhaps such approaches as that employed in PASCAL will point the way to successful description. In the meantime, CSC suggests that a standard HOL description will be the most successful if sufficient funding is provided and if the writers will give the same attention to clarity as has been given in the past to other disciplines. An approach to preparing such a specification must include sufficient review cycles and field trials to reach a satisfactory level of completeness. Furthermore, the language specification should be prepared in machine-readable form such that document editing facilities may be applied to accommodate changes, updates, and corrections quickly.

The standard reference document should address the problems of the implementation-defined semantics. There is no profit in restricting the scope of optimization for the arbitrary preference for a totally unambiguous specification. A sensible balance can be maintained between portability and implementation freedom that does not compromise the objectives of the LCF. The language standard and its resulting portability will be mechanized not by the language description, but by the compiler implementation tool. Therefore, expending effort in achieving profitless rigidity in the specification defeats the higher goals of the LCF and raises the price of compiler development at no gain to the community.

10.2 TUTORIAL TEXT

Inexperienced programmers new to the HOL will find the very thoroughness of the standard reference document too intimidating for the purposes of introductory familiarity with the language. CSC recommends that a student text be prepared that stresses fundamentals and includes numerous programming examples. The CS-4 text and the NCR Student COBOL text are useful models of the approach that might be taken to prepare such a document. In later stages of LCF development, the tutorial text could be supplemented by programmed instruction techniques employing video terminals connected to the LCF central site (or satellite sites) via the LCF network facilities. Once the student has mastered the fundamentals and has achieved a level of comfort with the language, he should be discouraged from using the tutorial text. To this end, the student text should provide ready cross-reference to the standard description. Thus, part of the learning process will be to gain mastery of the standard reference document.

10.3 TEACHING GUIDE

The principal tool in teaching the language will be the tutorial text described above. However, to supplement this, a teaching syllabus should be prepared which includes:

- Class schedules for courses aimed at beginning, intermediate, and advanced levels of instruction
- Classroom examples
- Classroom problems
- Examinations
- Organization of materials to stress the difficult language features
- Guide to machine usage of the HOL compiler, operating system interface, debugging tools, and program validation tools

The dual goal of formal HOL instruction should include mastery of all LCF tools as well as the HOL itself.

10.4 SUPPLEMENTARY MATERIALS

Useful supplements to the above language description materials include:

- Language reference handbook summary
- Program validator handbook summary

- Debugging tool handbook summary
- Compiler user's guide which presents the details of the proper use of the compiler for each LCF-supported host and target including such useful supplements as optimization hints, linkage conventions, interface with other languages, etc.

SECTION 11 - STATISTICS COLLECTION

The implementation of the statistics collection function within the LCF should be governed by the following basic functional design objectives:

- Static program statistics must be collected on a selective basis for individual programs
- Dynamic profile measurement must be accommodated on a selective basis
- The raw statistical data must be placed in a permanent repository from which it may be recalled by users and administrators or managers
- Execution measurement artifacts must be consistent with both the program validator and run-time debugger requirements, and all must share the same interface
- The production of statistical reports must be largely a postmortem function, although this does not exclude printing of compilation statistics for individual modules, where the function is imbedded in the compiler

11.1 STATIC PROGRAM STATISTICS

The most logical place in which to locate the statistics gathering function is within the HOL compiler. The main function of the static collector is the recognition of primitive language forms. Since this is entirely consonant with the syntax analysis function of the compiler, the collection function is implemented at minor additional performance cost in the compiler. The main element of the implementation is the insertion of counters indexed by language token. Using the JOCIT architecture as a base, this function may be imbedded almost entirely within the "precognition" or token recognizer routine. Of course, this function may be implemented by a stand-alone processor, but this approach duplicates much of the syntax analysis phases of the compiler and presents a language compatibility maintenance problem. Furthermore, as pointed out in Section 3, statistics on such items as object program generation or optimization must be collected by the compiler, so it seems reasonable to consolidate the functions there.

The principal implementation questions are the interface between the collector and the data repository and the form of the data repository. There are two alternatives for choosing the data repository. The first is to define a standard file (perhaps in the system library or at least one accessible to all accounts) which is implicitly known to the compiler. The output of all data collection would be expressed in tabular form and written as an update to the file. The second approach calls for the data repository to be part of the user's own account with access to it permitted from the statistics postmortem analyzer.

The problem with the second approach is that such a file is not protected from inadvertent corruption by the user. The first approach potentially shares this same failing, in that one would expect that the facility to update a system file grants a general access which would permit willful or accidental corruption. To some degree, this can be reduced through the use of special monitor calls within the compiler and perhaps the grant of some special privilege to the compiler program only which excludes update access for normal user programs. This is a somewhat simplistic approach, but it would serve the purpose of protecting the statistics data repository at a small cost in system implementation.

Yet another possibility, and one which does not require system modification, is to provide system data repository update routines in the form of library procedures whose code includes some means of verifying that the caller is one of the sanctioned updaters; trespassers would be rejected. This verification could take the form of passwords in the calling sequence. Of course, this is a good deal less secure than system level protection; the principal element of security is in not publishing the passwords and in keeping the identification of the data repository file name a secret as well. Such procedures are clearly not impenetrable, but they do guard effectively against inadvertent corruption.

There is no foolproof way to protect against mischievous interference, and since the cost to the LCF for the partial or total corruption of the data repository is not calamitous, elaborate security measures are probably inappropriate.

Both replacement and append modes of update are required. Static language usage data belongs to the first category. A new compilation of module A occasions the replacement of the static data record for module A within the data respository. Error data is more appropriately inserted as an addition to, rather than a replacement for, module A error data so that cumulative statistics may be gathered. Cumulative statistics for all data probably are unnecessary and only marginally useful; the file space required for the repository data base is already very large, and the cumulative gathering for all data might expand it beyond reasonable management.

A sample of statistical report is shown in Figure 11-1. It was produced by the DECSYSTEM-10 J73 compiler, and it shows summary counts for all the language forms suggested in the JLMT manual. This report was produced directly by the analysis phase of the compiler. A similar report would be generated by the postmortem analyzer suggested here for the LCF statistics-gathering facility.

The suggested categories of statistics to be collected and implemented within the compiler for the LCF are:

- Total number of tokens
- Total number of statements
- Total number of statements by statement class (declarative, processing, direct code, comments, directive)
- Total number of statements by distinct statement type (IF, GOTO, TABLE, !COPY, assignment, etc.)

```
JOVIAL  V.091075  3/23/76 16:56  MODULE:ACKER.          PAGE 1

,ACK=ACKER/IND/STAT

     1.   00200    PROC ACKER(MM,NN)U ;
     2.   00200         BEGIN "ACKER"
     2.   00350         ITEM MM U ;
     3.   00400         ITEM II U ;
     4.   00500         ITEM NN U ;
     5.   00600         TABLE VALUE(9)U ;
     6.   00700         TABLE PLACE(9)S ;
     7.   00800         OVERLAY VALUE:ACKER ;
     8.   00800         IF MM=0 ;
     9.   00900              VALUE(0)=NN+1 ;
    10.   00900         ELSE
    11.   01000              BEGIN
    11.   01100              VALUE(0)=1 ;
    12.   01200              PLACE(0)=0 ;
    13.   01200              LOOP:
    13.   01300              VALUE(0)=VALUE(0)+1 ;
    14.   01400              PLACE(0)=PLACE(0)+1 ;
    15.   01500              FOR II:0 BY 1 WHILE II<=MM-1 ;
    16.   01600                   BEGIN
    16.   01600                   IF PLACE(II)=1 ;
    17.   01700                        BEGIN
    17.   01800                        VALUE(II+1)=VALUE(II);
    18.   01900                        PLACE(II+1)=0 ;
    19.   01900                        IF II=MM-1 ;
    20.   02000                             GOTO CHECK ;
    21.   02100                        GOTO LOOP ;
    22.   02200                        END
    23.   02200                   IF PLACE(II)<>VALUE(II+1);
    24.   02300                        GOTO LOOP ;
    25.   02400                   VALUE(II+1)=VALUE(II);
    26.   02400                   PLACE(II+1)=PLACE(II+1)+1 ;
    27.   02400                   END "II"
    28.   02500              CHECK:
    28.   02500              IF NN<>PLACE(MM);
    29.   02500                   GOTO LOOP ;
    30.   02500              END
    31.   02500         END
```

Figure 11-1. Static Statistics Collection (1 of 3)

```
JOVIAL  V.091075  3/23/76 16:56  MODULE:ACKER.         PAGE 2

STATISTIC NAME                                     OCCURRENCES       PERCENTAGE

CHARACTERS                                              484
LINES                                                    26
SYMBOLS                                                 217
     KEY WORDS                                           28             12.90
          BEGIN                                           4             14.29
          BY                                              1              3.57
          ELSE                                            1              3.57
          END                                             4             14.29
          FOR                                             1              3.57
          GOTO                                            4             14.29
          IF                                              5             17.86
          ITEM                                            3             10.71
          OVERLAY                                         1              3.57
          PROC                                            1              3.57
          TABLE                                           2              7.14
          WHILE                                           1              3.57
     COMMENTS                                             2              0.92
     CONSTANTS                                           28             12.90
          INTEGER                                        28            100.0
     SIGNS                                              100             46.8
          +                                              10             10.0
          -                                               2              2.0
          =                                              12             12.0
          <=                                              1              1.0
          <>                                              2              2.0
          ,                                               1              1.0
          :                                               4              4.0
          ;                                              26             26.0
          (                                               1              1.0
          )                                               1              1.0
          [                                              20             20.0
          ]                                              20             20.0
     ABBREVIATIONS/DEFINE FORMAL PARAMETERS               6              2.76
          S                                               1             16.67
          U                                               5             83.33
     NAMES                                               53             24.42
          LABEL                                           6             11.32
          PROC                                            1              1.89
          TABLE                                          20             37.74
          SIMPLE-ITEM                                    22             41.51
DECLARATIONS                                              7
     SIMPLE-ITEM                                          3             42.86
          NON-BASED                                       3            100.0
     TABLE                                                2             28.57
     OVERLAY                                              1             14.29
     PROC                                                 1             14.29
STATEMENTS                                               32
     SIMPLE ASSIGNMENT                                    9             28.13
     FOR                                                  1              3.13
     IF                                                   5             15.63
```

Figure 11-1. Static Statistics Collection (2 of 3)

```
JOVIAL  V.091075  3/23/76 16:56  MODULE:ACKER,        PAGE 3

STATISTIC NAME                                        OCCURRENCES    PERCENTAGE

     GOTO                                                   4           12.50

PROGRAM SUMMARY
DATA/VARIABLES              000000 - 000031
INSTRUCTIONS/CONSTANTS      400000 - 400063
EXTERNS:  NONE
INTERNS: ACKER
FILES REFERENCED:   NONE

26 LINES   0 MESSAGES

ELAPSED TIME       2.287  SEC
```

Figure 11-1. Static Statistics Collection (3 of 3)

- Total number of data declarations by type (integer, real, etc.)
- Total number of processed constants by type
- Total number of subprograms, also segregated into procedure and function categories
- Total statistics on structure declarations (to include table packing class, type of table, number of dimensions, number of constituent fields, distinction by parallel or serial form, occurrence of multiword or word-spanning fields, etc.)
- Total number of COMPOOL definitions
- Total number of unreferenced declarations by class (procedure, table, item, whether COMPOOL or LOCAL, etc.)
- Total number of copy definitions
- Maximum procedure nesting level
- Counts by n-level procedure nesting (e.g., number of procedures at outer scope, number of procedures at first nesting level, number at second level, etc.)
- Maximum block nesting
- Counts by n-level nesting
- Maximum FOR statement nesting
- Counts by n-level FOR statement nesting
- Total error count and counts segregated by error number
- Error count by statement type and severity level
- Error count by procedure (including functions and closed routines)

- Count of optimizations performed by class (common expressions recognized, constant arithmetic performed, variables redistributed from loops, operations reduced in strength, dead stores deleted, common values hoisted, delayed stores inserted, etc.)
- Count of called procedures by number of references (i.e., number of procedures called zero times, once, twice, etc.)

The following object program and compilation statistics also would be collected:

- Object program data size, total and segregated by procedure
- Object program code size, total and segregated by procedure
- Number of registers used by procedure
- Number of REF items
- Number of DEF items
- Number of COMMON block references
- Size of required data space for each based procedure (relevant to J73)
- Count of library subroutine references by routine
- Count of generated instructions by instruction mnemonic
- Total, average, and percentage of generated instructions by statement class
- Total compile time
- Compile time per statement
- Compile time per token
- Compile time per source character

- Compile time per source character excluding comments
- Compile time per source character excluding comments and redundant spaces
- Compile time per statement by statement class
- Compile time per phase segregated by subfunction (e.g., total analysis time, time for generating source listing if requested, time for reformatting, time for assembly listing editing, etc.)
- Maximum space requirements by phase and subfunction
- Acquired and released space by phase
- Sizes of dynamically-acquired tables
- Sizes of intermediate files
- Cost of intermediate files expressed as the product of storage blocks and seconds of allocation
- Total number of input (source, copy, and COMPOOL) records read and average record size in bits (or words or bytes)
- Total number of output records written (updated source, object file, statistics file, printer file)

11.2 DYNAMIC PROFILE MEASUREMENT

It is proposed that the design for dynamic profile measurement be integrated with that discussed later in Section 13 for the instrumentation of performance analysis within the program validation context. In order to implement the timing component of the profile, two approaches suggest themselves. The first is implemented solely through the CDD (Composite Debugging Dictionary) interface. This dictionary, generated by the compiler, contains statement description information. It is possible for the code generator to provide information entries in the code file which contain total execution time for each statement. The EDITOR then includes this information in the CDD that it constructs. Then, execution measurement proceeds as follows:

- The object program is loaded, and breakpoints are implemented at the beginning of each basic block
- The program is executed, and the implanted breakpoints merely increment a counter; i.e., a running sum is maintained for each basic block
- Postmortem analysis processes the counters and constructs a histogram by basic block (or some other unit, if desired) which is the product of the count and the sum of the execution times for each statement in the block (obtained from the CDD); other reports also may be generated

This approach is entirely independent of the operating system and may be implemented for any target environment. The outputs of the run (i.e., the basic block counters) are placed in the data repository. The postmortem analysis may then be activated at intervals independent of the actual run. Of course, the programmer may invoke the analysis as part of his job if he so desires.

The second approach is more sophisticated and does not disturb the measured program with implanted instrumentation (although the first approach, which derives rather than measures the timing, does not skew the results because of the implanted instrumentation). The following suggestion makes certain assumptions about the operating system under which the user code executes and may not be considered a general solution. In this approach, the program to be measured executes as a "co-task." A timing program is run in parallel with the program to be measured. They execute as equal subtasks of the user task. The timed subtask executes for a prespecified interval, and then the timing subtask is invoked. The only function of the timing subtask is to record the program location counter for the timed task as it existed when it was interrupted for subtask slicing. In this way, the timing task obtains a pseudorandom sampling of the execution location of the timed task. This location counter data is stored in a table that is inserted at task termination into the data repository. Postmortem analysis may then be invoked to translate the sampled data into a histogram of execution profile according to any selectable granularity (statement, basic block, or even procedure).

This co-tasking approach is suggested to improve the randomness of the instruction counter sampling. If an entirely separate task were invoked to perform the sampling (which is effected by reading the saved location counter in the timed task's control block), the counter would reflect the state of the task when its time slice was interrupted which may be for a variety of reasons other than time-slice end (monitor call, I/O request, etc.) and thus not a truly representative sampling of the counter over a contiguous spectrum of the time of the object task's execution.

The implementation of the second approach requires that a subtasking capability be implemented in the operating system that permits micro-time-slicing within the task which is independent of the time-slicing which the system employs for intertask commutation. Furthermore, the intratask slicing must occur as if the timed subtask were executing continuously; i.e., the timing subtask is not activated on any event except that of micro-time-slice expiration. The micro-time-slice interval may be less than or greater than that of the macro interval; it merely needs to be chosen so as to provide sufficient points for constructing a meaningful histogram while imposing the least disturbance on the system. Clearly, a micro-slice much smaller than the macro-slice will greatly perturb the overall system performance through a deluge of clock interrupts.

In addition to histographic profiles, the basic block counters provide raw data for any number of statistical analyses, including:

- Single statement average execution time by statement type
- Frequency of execution by statement type (less important than actual execution time but useful when displayed against error statistics for the same statement type)
- Relative frequency of execution of parallel paths (data which may be used by the optimizer)
- Identification of unexecuted paths
- Timing and frequency of system and library routines

The primary purpose of the execution profile is to identify the most heavily travelled regions of the program under measurement. From the point of view of HOL appraisal, the profile may be used to identify unused language features, misused language features, and language features meriting optimization attention. For the user or program manager, the data is used in a program validation context as described in Section 13. In this case, recognizing the highest frequency program regions gives some guidance in overall program tuning and performance improvement.

While the first method proposed above is less costly than the second, it places some considerable burden on the code generator to compute the statement timings. For some machines, the timing is not easy to calculate and may be only approximate (within some acceptable limits of accuracy, of course). More importantly, this method precludes timing of system routines - the second method supports timing of system code executing as part of the task - or library routines for which statement timing data is unavailable and implantation is impractical.

11.3 STATISTICAL SUMMARIES

The raw data collected by either of the above approaches are sufficient to develop all of the reports suggested in the JLMT document discussed in Section 3. The basic histographic profile displays the granules of the program (statements, basic blocks, or procedures, for example) along the abscissa, and percentage of total execution time spent within the granule on the ordinate. These are, of course, derived differently in the two methods. The first collects the summary times for each granule directly; the second traps the location counter which must be correlated to the granule. In the first method, the times must be reduced to percentage frequency; in the second, the percentage is directly computed as the quotient of the individual counters divided by the total number of samples taken by the timing task.

In order to support both the statistics collection and program validation functions, the collected execution data must be appended to the data repository for the program under measurement. This permits both individual run and cumulative statistics analysis.

The reports generated are presented in, but not limited to, the following list:

- Basic histographic program profile
- Histograms of both static and dynamic usage of language features; the dynamic display will be by both frequency of usage and execution time
- Histograms of static and dynamic usage of library routines
- Histograms of static and dynamic usage of variables and constants by type
- Summaries of language definition classes: block-local, module-global, DEF, REF, copy-originated, COMPOOL-defined, and undefined

- Histograms of usage of primitive operators (+, =, AND, etc.)
- Histograms of usage of programmer-defined procedures and functions
- Histograms of time spent in FOR-loops and non-formal loops (i.e., those defined by backward branches)
- Histograms of programmer label references (dynamic) to aid in locating heavy GOTO usage for possible restructuring by CASE or FOR, for example
- Error histograms by language feature
- Program execution failure histograms by language feature and by particular program region
- Error histograms by error severity, class (processing, declaration, etc.), and error number
- Cumulative error statistics by individual module (error rate as a function of the number of times the module is compiled)
- Cumulative histograms of program failure by program granule (to aid in locating troublesome routines that are not necessarily a function of language usage difficulty)
- Compiler performance profiles showing size and speed of the compiler as functions of input data characteristics such as number of source characters or tokens or statements
- Graphs of object program size versus source program size
- Annotated source listings that give execution histograms alongside formatted source statement listings

11.4 <u>CONTROL OF THE STATISTICAL-GATHERING PROCESS</u>

It is suggested that two methods of invoking the statistics collection process be implemented. The first is user selection, in which the compiler is instructed both to create the CDD and, optionally, to collect the compilation statistics described earlier in this section. The user may at any time invoke the post-mortem analysis routines to create any of the available summaries and displays.

More important to the LCF function is the provision for LCF or programming managers to invoke statistics collection. The compiler, for example, can be made sensitive to flags in system tables to which access is provided either through direct inspection or via monitor calls. The flags would be set and reset - by on-line function or through the application of some periodic process - by the relevant managers. The managers could select entire projects or individual accounts or even individual programs for activation of the statistics-gathering function.

Execution monitoring can also be controlled in this dual fashion. It is of central importance that the statistics gathering tool be invocable by the system managers according to any sampling criterion they deem useful. While CSC endorses user selection as a supplement to this, it would limit the effectiveness of the process to grant invocation exclusively to the user.

SECTION 12 - COMPILER VALIDATION

As presently constructed, the JCVS tests provide an excellent base for the compiler validation tool to be incorporated into a LCF based on either (or both) of the JOVIAL/J-3 or JOVIAL/J73 languages. In order to realize their maximum potential, CSC recommends the enhancements described in the following subsections.

12.1 EXTENDED LANGUAGE FEATURE TESTING

The JCVS modules should be extended to include more exhaustive testing of the following language forms:

- Input/Output statements (J-3)
- ENCODE/DECODE statements (J-3)
- MONITOR statements (J-3)/! TRACE directive (J73)
- BIT, BYTE, and table entry assignment, exchange, and relationals
- Based procedures (J73)
- Procedure parameter passing (including passing formal parameters as actual parameters)
- Double precision operations

12.2 OPTIMIZATION TESTS

Test modules should be included to exercise the global optimization capabilities of the compiler. However, the tests are not conveniently made self-checking and would probably contain certain target machine or compiler logic dependencies. The best analysis technique is sight checking of the generated code and optimizer messages. The proper construction of comprehensive optimizer tests will include complex flow and a thorough exercise of procedure calls and associated spoil (or side effect) logic.

12.3 INTEGRATED FEATURE TESTING

Modules which combine language features and which are organized as independent modules utilizing COMPOOL/copy facilities should be added to the JCVS set. Tests should also include complex link-edits of compiled modules. Tests constructed to meet these requirements need not be self-checking, but should be thoroughly documented. The test documentation should specify program inputs and expected program outputs.

12.4 AUTOMATED TEST CONSTRUCTION

The automated test selection feature of the J-3 JCVS should be applied to the J73 JCVS. Furthermore, this facility should be expanded to permit rapid and convenient creation of tests in which the selection criteria is based on language features (much as the appendixes to the J73 JCVS User's Guide suggest). The test construction facility should be made available to the compiler implementation team to assist them in integration testing during compiler development. This would provide an invaluable aid to implementers and should add significantly to this activity.

12.5 JCVS USER'S MANUALS

The current level of JCVS documentation should be enhanced to provide more specific information on test construction. Furthermore, additional documentation on the nature and objectives of each test module should be included.

12.6 SUPPLEMENT TO JCVS

During the course of the JOCIT implementation, approximately 200 JOVIAL/J-3 tests were devised to aid in component and integration testing. These provide a useful supplement to the JCVS tests, but they require certain enhancements. The first requirement is to standardize the result reporting technique and to add self-checking logic where it is currently omitted. The tests themselves then must be subjected to rigorous evaluation before they can be certified as standard validation supplements. These tests - known as the J-series (disc-resident under GCOS) - are readily adaptable to other HOLs. Some 48 of these tests have been rewritten in J73 and currently are used as supplementary validation tests for the DECSYSTEM-10 and Hot-Bench J73/I compilers. The adaptability of the tests to new targets is facilitated by the use of DEFINE parameters for machine characteristics.

12.7 ACCEPTANCE PROCEDURES

The purpose of compiler validation is to provide measurement criteria by which compiler-buying agencies can objectively determine that the developed compiler meets the requirements of the language specification. The existence of standard, debugged JCVS tests is the main requirement in fulfilling this validation objective. Within the scope of the LCF, standard validation procedures which describe the tactics of the compiler measurement/acceptance activity should be prepared and published. These procedures would be applied for each rehosting and retargeting effort undertaken within the LCF community. The availability of these standards will reduce the cost of the activity and raise the level of objectivity as well. The standard validation procedures are the logical companion to the standard tests.

SECTION 13 - PROGRAM VALIDATION

The JAVS technology, discussed in Section 5, provides a suitable basis for program validation tools within the LCF. Assuming that the first implementation of an LCF most likely will be oriented towards the JOVIAL language in general and J-3 in particular, JAVS can be incorporated directly. Its availability in a J-3 LCF will provide field testing experience which will add an objective measure of its worth and give an opportunity to determine those aspects of the tool meriting enhancement or modification to broaden its applicability.

Since program validators do not yet exist as front-line production tools, the incorporation of JAVS into the LCF may be treated like something of an R&D venture. In the future, it may be the case that program validators will exist as *essential elements of the program production process*, but that clearly is not the case today. The employment of an untried technology such as JAVS really presents no risks either to the LCF sponsor or to the subscribers if treated as a potentially cost-reducing component meriting exposure in an R&D module. Perhaps certain pilot projects may be underwritten by LCF users which will specify JAVS-like validation as an integral component of the implementation.

It is recommended that JAVS be included in the standard tool repertoire for a J-3 LCF. The first level of such an LCF implementation should incorporate the tool as it is presently constructed. The potential shortcomings observed in Section 5 may then be exposed and examined, and field results may be analyzed to determine the kinds of enhancement required.

Subsequent levels of JAVS (or JAVS derivative) implementation must address certain deficiencies observed in the analysis of Sections 1 through 6. Furthermore, remaining faithful to the principle of integrated tool development requires some reassessment of what may be observed as redundancies with other LCF components. These ideas are discussed in the following subsections.

13.1 PROGRAM VALUE VERIFICATION

The value verification provisions of JAVS - the TRACE, EXPECT, and ASSERT directives - are more suitably implemented directly in the HOL. These are, for the J-3 language, translated by JAVS into MONITOR statements in any case. The JAVS TRACE directive is almost entirely redundant with the J-3 MONITOR statement, and since there is the provision in the JOCIT compiler for the optional suppression of MONITOR statement compilation, and since the output of TRACE - and EXPECT and ASSERT as well - is simply displayed and not collected with other performance data, there is no justification for the redundancy.

Two suggestions emerge from the above observations. First, the three directives should be implemented as extensions to the J-3 language. TRACE could be eliminated entirely, although there is some marginal convenience in its use versus MONITOR. For example, TRACE ON ABC $ ... TRACE OFF ABC $ is less clumsy to write than its MONITOR equivalent:

```
ITEM        T'ABC  B $
MONITOR (T'ABC)  ABC $ ...
T'ABC = 1 $    "EQUIVALENT TO 'TRACE ON ABC' " ...
T'ABC = 0 $    "EQUIVALENT TO 'TRACE OFF ABC' "
```

The second suggestion is to include ASSERT and EXPECT results as part of the JAVS library output during test measurement rather than simply displaying the results on the MONITOR list file. Postmortem analysis could then provide for annotations to the source listing - much as DD-path exercising is shown - to pinpoint the failed assertions and expectations.

13.2 INSTRUMENTATION

As pointed out in the JAVS analysis, the approach to instrumentation adds considerable overhead to the size and considerably reduces the speed of the compiled programs. The source level of instrumentation which JAVS employs must be considered an interim solution to a problem which, within the scope of an integrated LCF, demands a more encompassing solution. For the first level of LCF implementation - assuming JOVIAL/J-3 as the HOL - the present JAVS approach gives the LCF community a chance to evaluate the JAVS method of program validation. For future LCF versions, the integration of the JOCIT compiler, the JAVS methodology, and the support debugging tool may be expected to vastly strengthen the approach.

It is proposed that DD-path identification be included as supplementary information in the compiler-constructed CDD. Implanting breakpoints at each DD-path origin and convergence may be effected at the machine language level. At execution time, DD-path tracing is effected by implanting TSX instructions (or the equivalent) at the origins and convergences of the paths - the points are located through the CDD - which provide linkage to the run-time DD-path data gatherer. With this approach, the object module is not inflated in size, and the run-time cost of data collection is reduced as well. It is likely that only a very few instructions need to be executed to record the DD-path tracing; namely, to store the address contained in the TSX register and return. Subsequent data reduction of the count data then may be correlated with the DD-path descriptors in the CDD to permit the same level of source-language correlation provided for by the present JAVS.

This approach does not change the results, only the implementation technique; but the performance improvement so realized is of significant importance to both the user and the computer installation managers. It is even possible to conceive of a totally integrated approach to performance monitoring; i.e., the same approach that permits JAVS-like data gathering serves the statistics gathering function equally well. Any program resident within the LCF then is subject to monitoring by either the user or the installation managers.

13.3 SOURCE TEXT FORMATTING

The source text formatting provisions in JAVS also may be more conveniently implemented within the analysis phase of the compiler. The DECSYSTEM-10 J73/I compiler employs an integrated source level formatter as part of the token recognizer within the analysis phase. This formatter provides all of the functions of the JAVS formatter including DEFINE expansion (which, in J73/I, includes parameterized DEFINEs). This is as useful a place for the existence of the formatter as the JAVS program. It is perhaps more useful because the user must compile his program, and the optional reformatter imposes only modest overhead on the compiling function - less than the cost of invoking yet a separate processor for the purpose. Although this is a marginal concern, it does seem desirable to include reformatting as part of the standard JOCIT compiler rather than restrict its availability by providing it solely as an adjunct to the validation function. CSC does not recommend that the reformatting function be deleted from the current J-3 JAVS (JOCIT does not presently include a source reformatter) but rather that JAVS-derivative implementations in the future should defer this function to the compiler.

13.4 FLOW ANALYSIS

The DD-path-oriented flow analysis of JAVS is somewhat redundant with the flow-analysis capabilities of the JOCIT compiler. If code straightening is added as an enhancement to the present optimization capability, a good deal more than DD-path flow analysis will be provided. The compiler, then, can generate the raw data currently obtained in STEPs 1 and 5 of JAVS and insert it into the CDD for subsequent JAVS processing at practically no cost.

Viewed in this way, the JAVS function then no longer needs to employ source language processing at all in order to identify the DD-paths. Even the statement type data can (and should) be inserted in the CDD, which serves both JAVS and the statistics-collection functions as well. Section 11 states additional details of the requirements to support statistics-collection.

SECTION 14 - PROGRAM DEVELOPMENT/INTEGRATION TOOLS

While the first objective of the LCF is to provide centralized control over all aspects of HOL development, maintenance, usage, and growth, an equally important function is to serve the program development needs of the subscribers. Clearly the compiler is the principal instrument of this latter goal. Performance measurement tools support the language use and system development as well. In addition, more fundamental tools, while not necessarily oriented to a particular HOL, are essential to a thoroughly integrated LCF. The proposed tools are:

- Source module text editor
- Object program linking loader
- Interactive and postmortem debugger
- Source language translator(s)

An important design objective which distinguishes these tools from others proposed in this report, is that these support tools should be designed to be as independent of language as possible. While the compiler, and to a lesser degree, the program validator, are by definition language-specific, these supplementary tools (with the obvious exception of the translator) need not, and should not be. Different operating systems offer these tools to varying degrees of conformance with the proposed functional requirements. To the degree that the host system, on which the pilot implementation of the LCF is undertaken, lacks these requirements, there will emerge the need to design and implement the tools. A cursory analysis of four operating systems under which an implementation of the LCF might be considered reveals the degree to which existing tools meet the requirement. Table 14-1 summarizes this analysis. The four systems are:

- UNIVAC 1108/INFONET-CSTS
- DECSYSTEM-10 Timesharing
- HIS-6000/MULTICS
- HIS-6000/GCOS Timesharing

Table 14-1. Comparison of Existing System Support Tools

System	Text Editor	Linker	Debugging	Translators
INFONET/CSTS	E-	E	E-	I
DECSYSTEM-10	A	A	A+	N
MULTICS	I	E	E	N
GCOS	A	I	I	N

In the table, the codes have the following meaning: N - nonexistent; I - existent but inadequate; A - existent and adequate for a first-level implementation of the LCF; and E - excellent mating of the performance of the tool with the requirement.

It should be observed that the DECSYSTEM-10 offers a variety of text editors the composite of whose features would rate an "E." However, no single editor alone is really sufficient. MULTICS is deficient in providing no true line editing facilities at all. Only INFONET/CSTS and MULTICS have conceived of and implemented integrated linking and debugging facilities. CSTS is deficient in its debugging capability by failing to support any HOL but FORTRAN (JOVIAL may be supported to a large degree, but procedure name scope and based data are not accommodated). None of the systems provides language translators of the kind envisioned for the LCF, namely, the facility to translate to command/control-oriented HOLs (except for CSTS on which a SYMPL to J73/I translator is available).

14.1 SOURCE MODULE TEXT EDITING

The general requirement for source module text editing is obvious, and does not merit much discussion. A summary of the specific requirements is given in the following paragraphs.

14.1.1 EFFICIENCY

Text editors are one of the most heavily used components of any interactive system. In order to impose the minimum performance degradation on the system, the editor should be written for maximum internal efficiency in terms of speed, compactness, and optimized file access. For those systems supporting reentrant programming, the editor should be written as a shared program.

14.1.2 LINE EDITING FACILITIES

The editor should be designed to permit the creation and editing of keyed files. Lines, identified by key number, should be addressable, and line ranges should be permitted for all editing functions. Insertions should be convenient. A well-designed editor avoids annoying idiosyncracies such as that found in the DECSYSTEM-10 SOS editing program in which operating on an existing file without keys causes the file to be automatically numbered in increments of 100 with a maximum line number of 99900. Large files are accommodated by dividing the file into "pages" of 1000 lines each. Subsequent editing commands must include page number as well as line number specification, the former of which is clumsy, error-prone, and poorly documented. TECO and QEDX (the latter under MULTICS, the former under both DECSYSTEM-10 and MULTICS) do not accommodate line numbers at all. In some circumstances, DECSYSTEM-10 TECO even destroys line numbers in a keyed file. The GCOS timesharing editor manipulates line numbers as a special textual component of the source line. Only CSTS treats line numbers as true file keys and permits insertion by means of fractional line numbers (to a precision of $10^{-6}+1$).

14.1.3 SEARCH AND SUBSTITUTE

This requirement specifies that the editor will permit the location and replacement of textual strings.

14.1.4 CONTEXT EDITING

The editor should accommodate identification of lines by some contextual condition (e.g., change the string "ABC" to "ABC+1" in all lines containing "IF BB=0"). The contextual delimiting should be exclusive as well as inclusive. Multiple contexts also should be provided.

14.1.5 CURSOR CONTROL

The user should be able to control cursor position within the edited line (e.g., to the beginning of the line, to the end of the line, to any particular column position, to the beginning or end of any particular string, etc.).

14.1.6 "WILDCARD" CONSTRUCTIONS

The user should be able to specify strings in which certain characters are replaced by a wildcard character (i.e., one which is matched by any character). This permits such editing functions as: delete all strings "XX??" (where "?" represents the wildcard character). The user should be able to override editor-defaulted wildcard representation, and the editor should support single and multiple wildcard constructions (e.g., delete all strings "XX(?)", where "?" in this example represents a multiple wildcard character, would delete "XX(1)", "XX(A+B)", etc.).

14.1.7 EDITOR VARIABLES

It is often convenient to generate certain string constructions. To this end, the user should be able to define editor variables, assign values to them, and substitute or insert them as textual strings in appropriate contexts or to use their values as components of editing instructions. For example, %A="JULY 4 1976" would assign the quoted string to the editor variable %A. Then the command insert string %A at the end of lines 100-999 would cause the string "JULY 4 1976" to be inserted as directed. Similarly, %X=1 would assign the numeric value 1 to the editor variable %X; the variable may then be used any place in the editing syntax in which numeric values are appropriate.

14.1.8 EDITOR FUNCTIONS

It would be useful to permit the definition of editor functions, i.e., callable strings of editor commands that return a value which may be used in whatever context the function is referenced.

14.1.9 REPEATED CONSTRUCTIONS

Analogous to HOL loop statements, repeated editor constructions could be used to apply the same editing commands over a range of lines or contexts. In particular, references to functions from within repeated constructions may be useful in creating repeated text patterns which differ by some computable string element (e.g., to generate a string of comma-separated, monotonically increasing integers).

14.1.10 CONDITIONAL COMMANDS

Analogous to IF statements in HOLs, conditional commands could be employed to permit more tailored contextual analysis for the application of other editing commands.

14.1.11 TEXT MOVING AND COPYING

The user should have the facility to move and copy blocks of textual lines. The CUT and PASTE functions of the GCOS editor, for example, are unnecessarily elaborate commands to achieve what is essentially the "move" function.

14.1.12 USER INTERFACE

The editor command language should present an accessible and easily mastered user interface. TECO, for example, is quite a powerful editor, but its command syntax is sufficiently obscure to hinder accessibility to its more powerful features. The CSTS editor language is quite concise, but somewhat less rich, and thus more easily mastered. Brevity is important in editing for the basic functions, but the more complex might profitably borrow from HOL-like syntax to encourage familiarity.

14.2 OBJECT PROGRAM LINKING LOADER

There is virtually no contemporary operating system that does not offer some kind of linking loader. However, few of these have been implemented with the kind of careful attention to design, performance, and user interface that such a fundamental tool deserves. Large program system development, in which perhaps hundreds of modules are combined, demands the utmost in linking convenience. The design of the linker must be carefully integrated with all aspects of language processor support and must be especially sensitive to the characteristics of the file management system. More than one linker whose internal logic was efficiently conceived faltered in performance because of excessive file management demands. Furthermore the well designed linker supports the full range of languages for which compilers are provided within the LCF.

CSC suggests that the LCF implementation not make the assumption that, for example, there is no compelling need to replace the link editor just because programmers have managed to circumvent its primitive features for several years. Such an argument is analogous to the argument assembly language programmers have against the use of HOLs. The basic linker features described in the following paragraphs are the minimum requirements and are presented without distinguishing between two fundamental approaches to linking: static linking and dynamic loading. Almost all of the features outlined are equally applicable to either scheme, the choice of which depends upon other system design features largely independent of HOL support. Note that of the systems discussed above, only MULTICS offers dynamic loading as the proper linking service.

14.2.1 USER INTERFACE

Most linkers seem to have been designed from the inside out with little attention to the convenience of or respect for the user. To support the features listed herein, a command language of some kind is required. CSC suggests it obey a syntax similar to the operating system's command syntax for maximum user interface consistency, and that it have constructions to aid the programmer which are not overly verbose or cryptic. The more exotic features of the linker should be expressable in a straightforward way. CSC suggests, in particular, that discrete paths in an overlay tree be identifiable in the command syntax as unique "scopes" such that labels, equate names, and even module names are scope-protected and will not interfere with occurrences of the same name at outer or parallel scopes.

14.2.2 MODULE PLACEMENT

The user must be able to place object modules in the linked program in any particular order, or the user may let the linker choose the order.

14.2.3 AUTOMATIC MODULE RETRIEVAL/LIBRARY HIERARCHIES

The user must be able to define a library hierarchy to specify the order of library searching for automatic module retrieval. Automatic module retrieval is an essential element of good linker design. The retrieval is based on external symbols defined in the program for which control dictionary entries for the symbols appear in the object module. At a minimum, the linker will use undefined external symbols as search arguments for looking up the control dictionary entries in the user-specified library hierarchy, one module at a time, until the missing symbol is found (or until all modules in the hierarchy have been searched and the symbol is undefined). Of the systems discussed earlier, only CSTS and MULTICS have integrated in any way the design of object modules and libraries with the requirements of program loading and inter-module reference. GCOS, which borrows from the early IBSYS technology, does not provide for any automatic module retrieval within a user hierarchy, the definition of which it also does not support. DECSYSTEM-10 is only slightly advanced in permitting certain sanctioned language processors to set unique bits in the object module control dictionary which the linker uses as flags to trigger system library searches for undefined symbols. Unfortunately, it makes no provision for creating user library hierarchies and insists on explicit inclusion of all modules. IBM-370/OS/VS and its variants at least permit the user to name the files he wishes the linker to have access to during the linking process, but the structure of partitioned data sets limits the number of external symbols possible for each module to six: the module entry point and up to five "aliases," or auxiliary entry points.

14.2.4 OBJECT MODULE FORMATS/OBJECT FILE ORGANIZATION

When considering implementing a linker for the LCF, the choice must be made whether or not to support only the LCF HOLs or to support all HOLs supported by the system under which the LCF is implemented. If only the LCF HOLs are supported, then the object module format and object file organization may be tailored to integrated requirements and to realize efficiency. Relating to the previously discussed subject of module retrieval, the object file organization will best serve these needs if all external symbols are collected in a single file directory much as the index is organized in indexed-sequential files. This will result in much more efficient performance by reducing the number of file accesses required to resolve external symbols automatically. Such an approach is likely to be at odds with current operating system file design (with the exception of CSTS and MULTICS, which support these concepts to a considerable degree).

An even more radical approach to this problem is to design the object module formats in an entirely machine-independent way. IBM 370 standard object and load module formats are examples of straightforward design, much of which is actually machine-independent. Such an idea significantly enhances the retargetability and rehostability of the compiler since the object module formatter element of the compiler editing phase is written only once.

14.2.5 SYMBOL DEFINITION

The user often will find it convenient during large program development to link incomplete programs. To assist him, the linker should permit the user to define external symbols as part of the control input. For example, EQUATE ERROR$= JOVMOV would resolve references to the external symbol JOVMOV to the location computed for the external symbol ERROR$. Absolute equates also should be supported.

14.2.6 BOUNDARY ALIGNMENT

The user should be able to specify boundary alignment for any externally relocatable element of the link such as a module, control section, or common block. The boundary alignment could be expressed as an absolute address or as some function of the "next available" location, such as double-word alignment, next byte, or next page.

14.2.7 AUTOMATIC COMMON PLACEMENT

Very often, linkers place peculiar restrictions on the placement of common blocks. Aside from the ability to place them at his will, the user should be able to expect the linker to place them sensibly, i.e., at the highest point in the link tree to which only downward references result.

14.2.8 CLOSED TREE STRUCTURES

Most linkers support only simple tree structure overlays. This is clearly not sufficient for most large program organization efforts such as command/control systems or compilers. One should be able to load independently-higher segments of the tree without necessarily loading the intervening branches. For example, it should be possible to express the structure shown in Figure 14-1.

Thus, references to F would entail loading of C but not necessarily of either D or E; and references to G would not entail loading of either LEG 1 or LEG 2.

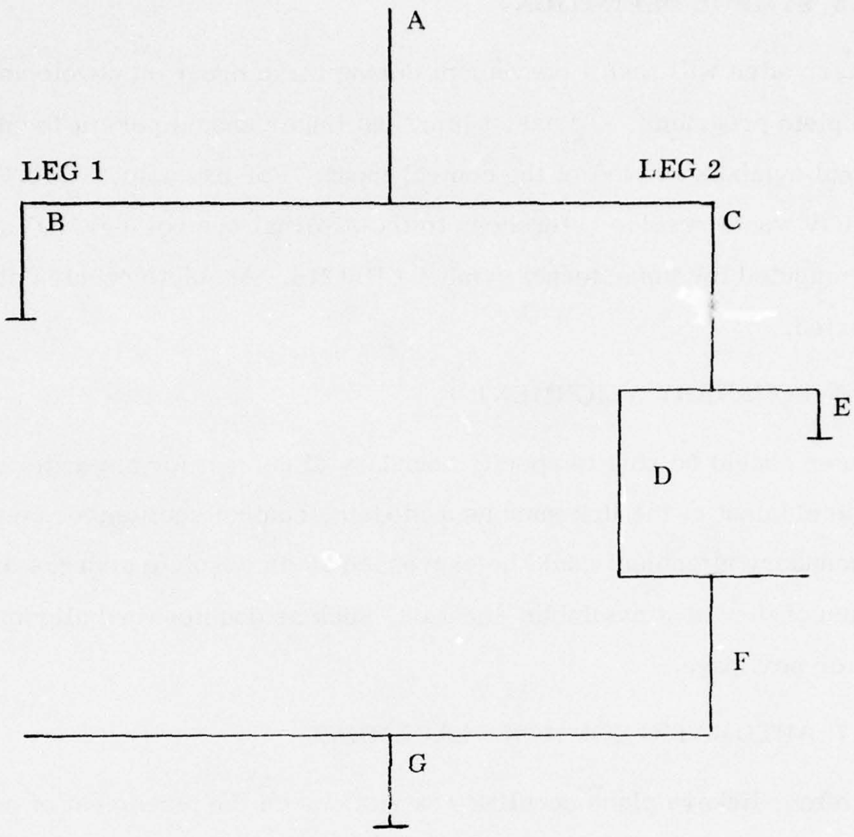


Figure 14-1. Sample Tree Structure

14.2.9 AUTOMATIC SEGMENT LOADING

The user should be able to specify which of the segments in the tree are to be automatically loaded when dynamic reference is made to symbols defined within them. Furthermore, to support the closed tree concept, the user should be able to define a path in a parallel subtree, which would entail automatic loading. In the previous example, the user might specify that when loading C, D and F are, in fact, automatically loaded; or that C is loaded alone and references to F automatically load E.

14.2.10 LISTING OUTPUT

Listing output by the linker should be optional and include a concise and readable memory map, cross-reference listing of external symbols by module reference, and comprehensive diagnostics.

14.2.11 COMBINED CDD

To support statistics collection, program validation, and debugging, the linker should be able to process the CDDs from each module and combine them into a single CDD for the linked program. The combined CDD would be a part of the loadable module constructed by the linker and thus facilitate the functioning of these other tools. Furthermore, the CDD ideally should not be core-resident (it is too extensive for that) but should be readily accessible through an integrated file management interface. To this end, the symbols in the CDD should be combined into a single hash-searchable segment which functions as an index into the remainder of the CDD. This makes reference to the CDD much more efficient during interactive debugging.

14.2.12 PARTIAL LINKING

To facilitate construction of large programs, it should be possible to link portions of the program independently, form them into partial link modules, and present them as input into a larger link activity. This idea is analogous to independent module compilation. To support this, the linker will construct linked modules in a form which is based on the relocatable object module format so that the linker can accept its own output as input once again. Furthermore, the user needs to be able to specify in partial linking which external symbol definitions within the link are to be retained as external symbols for resolution of references from outside the link. This approach makes a much more efficient use of the system resources by not requiring a total relink when only a single module or a few modules within an overlay leg are recompiled.

14.2.13 MULTIPLE ELEMENT PLACEMENT

Most linkers object to the appearance of the same module, control section, or common block in more than one place in the link tree. This is somewhat sensible in that it is difficult to resolve references which appear to be to multi-defined symbols. However, such resolution is not impossible, for example, when the multiple placement occurs in mutually-exclusive subtrees; the linker simply resolves references only to paths that are accessible to the reference and diagnoses all others. Such a facility is only implementable for explicitly placed modules; automatic placement by the linker is a contradictory function.

14.3 DEBUGGING

To support user program development, CSC proposes that the LCF provide a comprehensive debugging facility. For maximum utility, the debugging package should operate in both batch and interactive modes. The success of the debugging facility will be dependent upon the degree to which the compiler, program validator, and linker are integrated with it. The primary interface among these components will be the CDD created by the compiler and used by the other tools.

14.3.1 DEBUGGING FACILITIES

The following paragraphs discuss the functional requirements of the debugger; an outline of the requirements for the CDD follows.

14.3.1.1 Statement Implants

The user often wishes to place debugging statements at selected points in his program. These points should be identified by either program label or statement number. Because well-structured programs contain few, if any, labels, the statement number option is quite important in terms of the user interface. Statement numbers will be generated by the compiler and placed on the source listing of the compiled program for the user's reference. These implants will allow the full range of possible debugging statements (described in the sequel) including breakpoints. In addition, implants should be allowed in which the user can execute certain system commands of the kind that do not destroy the current program core image (such as displaying time of day, for example). Breakpoints will cause program interruption; in interactive mode, control is given to the user's terminals; in batch mode, input is then taken from the primary data input file. Return from breakpoints will permit the program to continue from the state of interruption. Implants also should be permitted in segments that are not necessarily loaded at the time the implant command is given. For example, such a facility is not permitted in DECSYSTEM-10 timesharing debugging.

14.3.1.2 Debugging Expressions

The user should be able to construct arithmetic and logical expressions that include program variables referenced by name, subscripting expressions consistent with declared variables, and constants of all forms including status constants, real constants, bit-pattern and character constants, and integer constants. Logical expressions should include the connectives AND, OR, XOR, and the unary NOT operator. Relational expressions which yield the values 1 for true and 0 for false may be constructed. IF statements which include a THEN option and an ELSE option should be permitted. Expressions and statements should be constructable in a form mirroring the HOL. Debugging sequences may include labels, GOTOs and repeated constructions (DOs). The following examples exhibit many of the required features:

> Example 1. IF ABC (II, JJ)<>0(SHOW ABC(II, JJ)) ELSE (RESUME); where the underlined words are debugging verbs, and all other forms are HOL-derived. Parentheses delimit scope for the "then" part of the IF and also for the "else" part. SHOW is used as the display verb, and RESUME indicates that execution of the program is to continue from the point of last interruption.

Example 2. DO .K=1,10(SHOW XX(.K)); where DO is the verb for repeated constructions, the repeated statements being contained within the following parentheses. The do-variable is .K - the "." is used to delimit debugger variables and distinguish them from program variables - and the limits are 1 and 10. Any number of statements - separated by semicolons - may appear within the parentheses. The expression XX (.K) refers to the dimensioned program variable XX subscripted by the debugging variable .K.

Example 3. .I=104; L1:SHOW, S #.I; .I=.I+1; IF .I≠114 (GOTO L1); where .I is initialized to 104, L1 is a label, the "S" suffix to SHOW says to display the source statement at statement number contained in .I, .I is incremented by 1, and the loop terminates with a test on .I and a GOTO. The effect of the example is to display 10 source statements beginning with statement 104. The "#" character is used to denote statement number.

Example 4. AT LABX (IF COUNT < 100 (TRAPS XYZ)); where the AT is used to place an implant at label LABX, the implant is the IF statement contained within the parentheses, and the TRAPS verb says to store-trap the variable XYZ (see sequel for trapping discussion). The effect of the AT statement is to insert the IF statement at label LABX. Thus, at LABX, when the program variable "COUNT" exceeds the value 100, the TRAPS statement is activated.

14.3.1.3 Data Display

The user should be able to display any program data in a form consistent with the declared data type. This includes the ability to generate table dumps which include all constituent items which are themselves displayed by type. Dumps of entire blocks are also required in which all constituent tables and items are displayed by type. Type display should include numeric, character, bit-pattern, and status.

14.3.1.4 Name Qualification

To resolve name ambiguity, the user must be able to qualify references to program variables. Qualification should be permitted by compilation module, named overlay segment, procedure name, or any other HOL structure in which qualification is permitted. The user also needs to be able to identify an implicit qualifier to be used when none is explicitly given with a variable reference. Thus QUAL A.B.C, for example, would establish the three-part qualifier as a prefix to all name references until the QUAL statement is cancelled or overridden. Similarly, the user should be able to name a qualifier abbreviation, such as QUAL X=A.B.C, in which "X" is used as a shorthand qualifier for A.B.C.

14.3.1.5 Debugger Variables

The user should be able to set and use debugger variables. These will require a delimiting syntax to distinguish them from program variables (for example, the use of the preceding "." character as used in the earlier examples). Debugger variables may be used anywhere their type is appropriate in the syntax of debugging statements.

14.3.1.6 Subroutine Calls

A useful debugging feature is the facility to call program procedures (and functions) from the debugging console; control returns to the console at subprogram exit. DECSYSTEM-10 timesharing permits this through the ability to execute machine code directly from the console. In addition to this, the user should be able to pass parameters in an HOL-like notation to which the debugger responds by constructing a system standard calling sequence. Thus,

CALL P1(X,Y,Z);

would invoke procedure P1 with parameters X,Y, and Z. Such a facility is especially useful in invoking program-specific debugging routines written by the user.

14.3.1.7 Trapping

One of the most difficult errors to identify is the case of the random store. This problem is intensified in languages supporting based storage. To assist in this problem, the user should be able to identify program variables or absolute addresses into which stores are trapped causing control to be passed immediately to the user's console. The most foolproof implementation of this requires essentially full interpretive running of the user's program. Since this is a relatively rare problem, the resultant degradation in performance may be acceptable. However, to mitigate this performance loss, the user should be able to specify a "trap granularity" of either statement, basic block, or other arbitrary designation. Thus, when basic block granularity is selected, the debugger takes control at block start and block end only and examines the value of the trap cell or cells (a range is permitted) at block exit. This eliminates full interpretive execution which may subsequently be employed to pinpoint the errant store. This feature should accommodate both load and store traps.

14.3.1.8 Core Dumps

Core dumps should be provided in which the dump includes both relative and absolute addresses. The user should be able to specify particular modules or control sections for dumping, and he should be able to distinguish by storage class as well (e.g., data segments only). Core dumps should include a display of module, control section, and block names.

14.3.1.9 Walkback

The debugger should provide the facility to display the chain of procedure calls, with parameters, between the current point of execution back to the original program invocation by the system (or back through a user-specified number of levels). This places a requirement on the compiler to generate subroutine linkages that conform to walkback requirements.

14.3.1.10 ON Conditions

The user should be able to specify machine exceptions that cause control to pass to the user's console. The ON conditions may be applied to the whole program or to specified domains within the program.

14.3.1.11 Program Inserts/Patches

The user must be able to insert program modifications expressed in machine-language mnemonic form with symbolic addresses. The debugger should provide an implicit patch area so that the user need not set aside a blank area for this purpose.

14.3.1.12 Source Statement Reproduction

The useful capability is reconstruction of the HOL source for any statement in the program.

14.3.2 COMPOSITE DEBUGGING DICTIONARY (CDD)

The CDD will be created by the compiler and output as part of the object module. The principal elements of the CDD are the symbol table, the statement table, and the flow table, discussed in the following paragraphs.

14.3.2.1 Symbol Table

The symbol table is constructed largely as a copy of the compiler's own internal symbol table. The symbol table contains both data names and attributes. The hierarchical data structure and block/procedure scope will be retained in the table to support name qualification. Only named data appear in the symbol table; compiler-generated temporaries, for example, are excluded. Names are retained with the attribute entries. It is the linker's responsibility to construct a combined CDD for a linked program.

The combined CDD is designed to be readily searchable by argument name - by separating the names and collecting them as a file index - with pointers to the attribute information within the combined CDD. At debugging time, then, it is possible for the debugger to hold a certain number of index pages in main memory and thus reduce the file management demands of the debugger. Attributive information includes data type, size, bit allocation of datum, parent structure, etc. Status constant names also are held in the symbol table, so that displays of status variables are appropriately mnemonic.

14.3.2.2 Statement Table

The statement table is indexed by statement number and provides the relative address within the program text of the first executable instruction of the statement. In addition, each statement table entry contains a pointer to the reformatted source module if necessary to permit source statement reconstruction. Label entries in the symbol table segment of the CDD also contain a statement number in order to provide access to the statement table during debugging.

14.3.2.3 Flow Table

The flow table provides an encoded representation of the program flow as a directed graph expressed in terms of basic blocks (vertices) and branch structure (edges). The beginning of each basic block is identified by statement number, thus providing a cross-reference to the statement table. The flow table also contains DD-path identification to support the JAVS requirements.

14.4 LANGUAGE TRANSLATORS

The functional specifications for the language translators must be expressed in terms of the specific language pair. Until the HOL for the LCF is chosen, it is somewhat pointless to attempt to write the specifications. However, because momentum is developing within the Air Force to move away from J-3 and towards J73, the LCF might commission the preparation of a J-3 to J73 translator. Because J73 derives heavily from J-3, this translation problem is not all that difficult. Even J-3 I/O statements may be translated as procedure calls to library-defined I/O subroutines. The file declaration itself may be translated conveniently into a J73 table, which may be passed as a parameter to the I/O subroutines. The remainder of the language translates rather handily.

Translation from other languages, such as PL/I, for example, may be expected to present a variety of problems. Some languages may permit use of special characters within identifiers that might not be easily accommodated in the target language. It may also be required to generate names during the translation, and this, of course, presents the problem of potentially duplicate definitions. Name qualification presents another problem that may require clumsy name construction with its attendant duplication possibility. Complex editing functions - such as exist in COBOL and PL/I - are resolvable at some level by the use of subroutine calls, but even this requires a complex translator. The problem of decimal arithmetic has been discussed earlier.

In summary, it may be supposed that translating between two languages on the same machine/operating system will present fewer problems than translating between two languages on two different machines. The advantages of the same machine are (1) that word size and data type problems are obviated, and (2) that the operating system already supports idiosyncratic language constructs - such as PL/I ON statements - such that library routines may be used to effect these functions that are not directly implementable in the target language.

The efficiency of the translation process itself is not of very much concern. Even where large volumes of programs will be translated, it is seen as a one-shot process, such that a low development cost appears to be the overriding design criterion. The translator should obviously be written in an HOL, preferably one that is oriented towards text manipulation problems. (The SYMPL to J73/I translator on the 1108 was written in six man-weeks using BASIC. The development time was quite short, but the translation was quite slow owing to the choice of BASIC.)

Inasmuch as source reformatting is suggested as an integral feature of the LCF's compiler, it would seem to be redundant to include target language formatting as a requirement of the translators. The translators need not be elegant; rather, they should be thorough and failure-resistant above all else. Even 100 percent translation is not strictly required, and diagnostics may be employed by the translator to aid the user in subsequent manual refinement of the translation.

APPENDIX A - COMPILER OPTIMIZATION ENHANCEMENTS

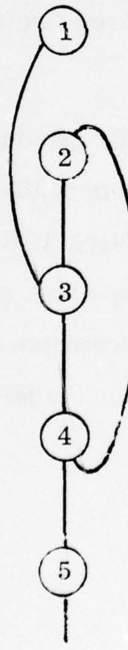
This appendix summarizes proposed enhancements to the optimizations presently performed in the JOCIT J-3 compiler. The optimization scheme now employed is a Level-I LNRA (Linear Nested Region Analyzer) scheme implemented as a two-pass process. Pass 1 (known as OPT1) performs all flow analysis and builds tables defining set and used information for each procedure and loop. Pass 2 (known as OPT2) performs the actual transformations to the code to realize the principal optimizations of common expression elimination, constant arithmetic, code redistribution, and operator strength reduction. The optimizer phases operate on the IL file generated by the analysis phases of the compiler; all transformations are expressed in the IL, and the output is also in the form of an IL file which is subsequently processed by the code generator phase (COGEN). This design permits selectable optimization. If the optimization phases are bypassed as they are when the NOPT option is selected, the IL generated by the front end is passed directly to the code generator. In this mode, only those local optimizations performed by COGEN are effected, and no global optimiation is performed at all.

The enhancements discussed in this appendix assume a solution entirely within the framework of the LNRA design. The sum of these improvements will be to raise the level of generated code significantly. It is expected that for any target machine the resultant code will occupy less space and execute in less time. Furthermore, these proposed enhancements do not violate the present target-machine-independent design, and thus the tool concept is not compromised in any way.

A. 1. 2 CODE STRAIGHTENING

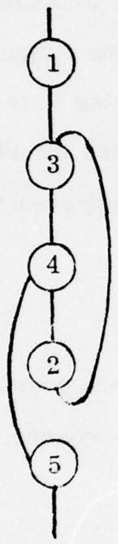
The optimization approach in the JOCIT Model uses the LNRA method in which program flow is determined in a single forward scan of the program (performed by Optimizer Pass 1, or OPT1). It is the assumption in this approach that a loop is formed whenever a label is reached via a backward branch. Furthermore, loop optimization is suppressed whenever it is observed that a forward branch enters a loop; i.e., redistribution and strength reduction are not attempted on multiple-entry loops (because of the complexity of placing the redistributed and strength-reduced initialization computations in each of the loop's entry blocks). These assumptions are entirely valid on well-structured or "straight" programs. However, they cause the optimizer to miss some cases when the code is not straight. A small program segment demonstrates this point.

The original order of the segment consists of the following five blocks and their interconnecting flow paths:



The optimizer sees 2-3-4 as constituting a loop formed by the backward branch from 4 to 2. However, the forward branch from 1 to 3 defeats the potential loop optimization as described above. Earnest, et al., in "Analysis of Graphs by Ordering of Nodes," JACM, Vol. 19, No. 1, January 1972, pp. 23-42, have proposed an algorithm that may be applied to place the blocks of a program in the straightest possible order.

Its application of the preceding example produces an ordering as follows:



This reordered segment is now a single entry loop (3-4-2), and redistribution and strength reduction algorithms may be applied to Blocks 3 and 4. Block 2, which is conditional, is excluded. The straightening, then, can be seen to have improved the optimization potential.

The code straightening algorithm of Earnest, et al., discovers all program loops in addition to straightening the order. As a result, the code straightener may be used to replace OPT1. Instead of reading the IL, the straightener will read an extended Global Names List (GNL) file, which will be called the Flow Graph File (FGF). This file will contain each program label, branch, start PROC and end PROC delimiters, PROC call, index switch label list, and index switch call. This information is sufficient to identify the basic blocks and interconnecting edges. This representation will then be reordered by the straightener, and the resulting straight order may be represented by an ordered table of "hatchecks" which identify the blocks of the IL to be read in order by OPT2. The FGF will be enhanced further to contain entries for each redefined variable--LHS of assignments, actual value output parameters, and name parameters--such that all redefined variable lists currently produced by OPT1 may be produced by the straightener. Since the FGF is considerably smaller than the IL, it is anticipated that the straightener will be significant'y faster than the present OTP1.

A.2 DEAD VARIABLE ANALYSIS

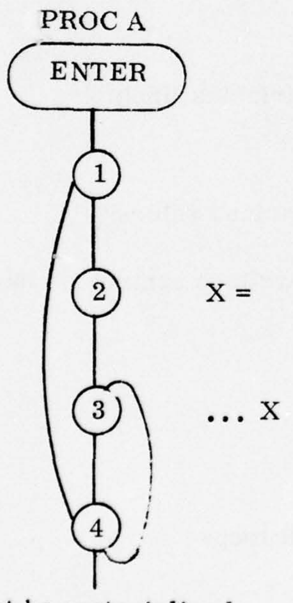
A program variable is said to be dead between its last reference and a subsequent definition. For example, in the program sequence

```
I  = ...
QQ = F(I)$
...
I  = ...
```

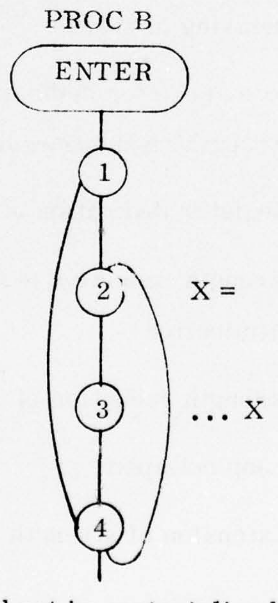
the variable I is dead between the assignment to QQ and redefinition of I in the last line. Recognition of dead variables raises two optimization possibilities which CSC proposes to examine for cost-effective implementation: (1) store suppression and (2) reuse of dead variable space.

In the above example, the original store into "I" may be suppressed if it is possible to retain "I" in some register between definitions. Following from this, it is seen that retaining "I" in a register means that no storage is required, and the allocated space for "I" may be reused during the program segment where "I" is "dead" by another program variable or compiler-generated temporary.

Dead variable analysis may be conveniently performed by OPT2 after an entire region has been processed. An extension to the definition of a dead variable may include that program segment between a last use and a program exit (or PROC RETURN). However, this may only be for those variables whose first reference in any PROC is a redefinition rather than a reference, i.e., those whose values do not survive from one invocation of the PROC to the next. Since JOVIAL does not permit the programmer to distinguish explicitly between these types of variables, this will be the compiler's task. The analysis procedure is not simple, as the following two procedures demonstrate:



X must be materialized at PROC exit



X need not be materialized at PROC exit

In PROC A there is a path (1-4-3) on which "X" is used before it is set, while in PROC B there is no path to the use of "X" in 3 which does not first redefine "X".

The objectives of dead variable analysis are:

- To suppress unnecessary stores
- To recognize possibilities of allocated space-sharing between programmer variables and compiler-generated temporaries
- To eliminate unnecessary storage allocation for variable held in registers
- To help the code generator retain variables in registers

A.3 LOOP OPTIMIZATIONS

In addition to redistribution and strength reduction optimizations currently performed, loop code can be improved in the following areas:

- Delaying of stores
- Index register dedication of loop control variables (including strength reduction-generated ones)
- Register dedication of redistributed and common values
- Strength reduction test replacement and dead loop control variable elimination
- Strength reduction of addition
- Loop collapse
- Extension of strength reduction to non-FOR loops

A.3.1 DELAYING STORES

Often with loop code, a variable is repetitively assigned. All but the store immediately preceding loop exit are redundant, and dedicating the variable to a register within the loop and delaying the store into the variable until after loop termination can improve loop performance. For example, in the following program:

```
YY($0$) = 0$
    FOR I = 1,2,99$
        IF XX($I$) GR YY($0$)$
            YY($0$) = XX($I$)$
```

if "YY" is dense-packed, the redundant stores within the loop may be quite costly. The optimizer may recognize the case and transform the program as follows:

```
temp = 0$
    FOR I = 1,1,99$
        IF XX($I$) GR temp$
            temp = XX($I$)$
                YY($0$) = temp$
```

Thus, in the example, if "temp" is dedicated to a register, both code space and execution time are reduced.

A.3.2 INDEX REGISTER-DEDICATION OF LOOP VARIABLES

Allocation of loop control variables to index registers eliminates loads and stores within the body of the loop, thus compressing the loop and speeding it up as well. This optimization should be applied both to programmer loop variables and to optimizer-generated loop variables arising from strength reduction. Such dedication may be expressed by means of accenting the

REPL IL operator to indicate that the LHS (the loop variable initialization and increment code, for example) is a candidate for loop dedication. The ENDL operator would signal the code generator to free such loop-dedicated variables.

A.3.3 REGISTER-DEDICATION OF REDISTRIBUTED VALUES

The optimizer moves all redistributed values into the loop entry block. This redistribution is indicated by the VALD operator. The optimizer will mark such redistributed values to make the code generator aware of the motion and to identify the loop from which the values were removed; this will enable the code generator intelligently to select which are the best candidates for register dedication. Even on limited register machines, such as the HIS-6000 series, this can be useful as in the case of a table search, for example:

```
FOR I = 0,1,999$
    IF XX($I$) EQ PATTERN$
        XX($I$) = 0$
```

In this case, PATTERN may be profitably assigned to an accumulator before the loop (especially helpful if XX is full-word addressable), and the interior of the loop is made smaller and faster. At the current level of optimization, XX(I) is loaded and compared with PATTERN, whereas (assuming XX is full-word addressable) PATTERN may be loaded outside the loop, and only the comparison code is required inside. This same optimization may be performed for values found common and therefore computed outside the loop.

A.4 STRENGTH REDUCTION TEST REPLACEMENT AND DEAD LOOP CONTROL VARIABLE ELIMINATION

During the process of strength reduction, it may be the case that all uses of a loop control variable will have been reduced, such that the loop control variable may be considered dead. In such a case, all references within the body of the loop will have been replaced by generated loop control variables, and the code to initialize, step, and test the original variable is all that remains. Strength reduction test replacement means to replace the test of the original loop control variable with a derived test on a generated loop control variable. This can be seen from the following simple example

```
FOR I = 0,1,9$
    XX($I*3$) = 0 $
```

which when reduced in strength by the optimizer effectively becomes:

```
temp = 0 $
   FOR I = 0,1,9$
       BEGIN "I"
           XX($temp$) = 0 $
               temp = temp+3$
                   END  "I"
```

If the test against I were replaced by a test against temp, the program could be rewritten:

```
FOR temp = 0,3,27$
    FOR I = 0,1$
        XX($temp$) = 0 $
```

Thus, the use of I is entirely dead, all references are eliminated, and the following simplified and improved program emerges:

```
FOR temp = 0,3,27$
    XX($temp$) = 0$
```

A.4.1 STRENGTH REDUCTION OF ADDITION

The current strength reduction algorithm includes only the reduction of multiplication and exponentiation. The reduction of addition (which reduces to another addition) is sensible when it leads to further reduction possibilities. For example, the following program,

```
FOR I = 0,1,99$
    BEGIN "I"
        FOR J = 0,1,99$
            AA($I,J$) = 0$
                END   "I"
```

in the current JOCIT model reduces only the implicit multiply of J; the subscript expression I, J is linearized to $I+d_1*J$, where d_1 is the first dimension of AA. Assuming that AA is 100 by 100 (d_1 is then 100) the equivalent code after strength reduction is:

```
FOR I = 0,1,99$
    BEGIN "I"
        t1 = 0$ "REDUCTION OF 100*J WHICH IS INITIALLY 0"
            FOR J = 0,1,99$
                BEGIN "J"
                    AA($I+t1$) = 0$
                        t1 = t1+100$
                            END   "J"
                                END "I"
```

An improvement to this would result from the reduction of the I+t in the inner loop. A straightforward reduction would give:

```
FOR I = 0 ,1, 99$
  BEGIN "I"
      $t_1$ = 0 $
          $t_2$ = I$ "FROM REDUCTION OF I+t WHICH IS
              INITIALLY I"
              FOR J = 0, 1, 99$
                  BEGIN "J"
                      AA($$t_2$$) = 0$
                          $t_1 = t_1 + 100$$
                              $t_2 = t_2 + 100$$
                                  END  "J"
                                      END  "I"
```

This reduction as shown is actually a degradation of the original program unless the dead loop control analysis is applied along with test replacement. The result is a significant improvement as the following equivalent program shows.

```
FOR I = 0,1,99$
  BEGIN "I"
      FOR $t_2$ = I,100,9900+I$
          AA($$t_2$$) = 0 $
```

NOTE: The expression 9900+I is loop constant over the inner loop, and thus is properly redistributed.

A4.2 LOOP COLLAPSE

If the preceding example were rewritten with the subscripts reversed,

```
FOR I = 0 ,1, 99$
  BEGIN "I"
      FOR J = 0 ,1, 99$
          AA($J,I$) = 0$          "J,I instead of I, J"
              END "I"
```

the effective reduction looks like the following:

```
t_3 = 99$
    FOR t_1 = 0,100,9900$
        BEGIN "t_1"
            FOR t_2 = t_1,1,t_3$
                AA($t_2$) = 0$
                    t_3 = t_3+100$
                        END  "t_1"
```

A close analysis of the above reduction reveals that the inner loop control variable (the generated one, t_2) steps consecutively from 0 to 9999; thus, the inner loop may be collapsed into the outer loop leaving a single loop as follows:

```
FOR t_4 = 0,1,9999$
  AA($t_4$) = 0$
```

This continuous stepping function may be recognized by observing that the difference between the terminal value of one iteration and the initial value of the next is precisely the step value of the inner loop control. For example, the terminal value of the t_2 on the first iteration is 99 (t_3), and the initial value of the second iteration is 100 (stepped value of t_1); the difference, 1, is the step value for t_2.

A comparison of the various levels of optimization discussed as applied to the simple example discussed above shows the progressive improvements. The examples shown assume the HIS-6000 as the target machine.

No Optimization

Total: 14/Inner loop: 8 (incl. multiply)

```
        STZ     I
L1      STZ     J
L2      LKQ     J
        MPY     100,DL
        ADQ     I
        STZ     AA,QL
        AOS     J
        LDA     J
        CMPA    100,DL
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 1: Current JOCIT Optimizer

Total: 15/Inner loop: 8 (no multiply)

```
        STZ     I
L1      STZ     J
        STZ     t1          t1 = J*100 = 0
L2      LDQ     I
        ADQ     t1          I+J*100
        STZ     AA,QL
        LDQ     100,DL
        ASQ     t1          t1 = t1+100
        AOS     J
        CMPA    100,DL
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 2: Reduction of Addition, Test Replacement, Dead Loop Variable Elimination

Total: 15/Inner Loop: 6

```
        STZ     I
L1      LKQ     I
        STQ     t2          t2 = I
        ADQ     9901,DL
        STQ     t3          t3 = 9901+I
L2      LDQ     t2
        STZ     AA,QL       AA($t2$) = 0$
        ADQ     100,DL
        STQ     t2          t2 = t2+100
        CMPQ    t3
        TMI     L2
        AOS     I
        LDA     I
        CMPA    100,DL
        TMI     L1
```

Optimization - Level 3: Level 2 + Register Dedication

Total: 11/Inner Loop: 4

```
        LXL1    0,DL        I = 0
L1      EAX2    0,X1        t2 = I
        EAA     9901,X1
        STA     t3          t3 = 9901+I
L2      STZ     AA,X2       AA($t2$) = 0$
        EAX2    100,X2      t2 = t2+100
        CMPX2   t3          test t2 vs. t3
        TMI     L2
        EAX1    1,X1        I = I+1
        CMPX1   10,DU
        TMI     L1
```

Optimization - Level 4: Level 3 + Loop Collapse

Total: 5/Inner Loop: 4

```
        EAX1    0,DU
L1      STZ     AA,X1
        EAX1    1,X1
        CMPX1   9999,DU
        TMI     L1
```

A.4.3 EXTENSION OF STRENGTH REDUCTION TO NON-FOR LOOPS

In the current JOCIT optimizer, strength reduction is applied only to formal (i.e., FOR) loops. Strength reduction may, however, be generalized to include the reduction of functions of any variable satisfying the following conditions:

- The variable is iteratively redefined once only with the scope of some loop; i.e., the variable ("I" for example) is defined by I = I+k or I = I-k
- The variable is nowhere else redefined within the loop
- The redefinition expression (k in the above example) is constant over the same loop in which the variable is iteratively redefined

The recognition of such variables, functions of which are candidates for reduction, will be the responsibility of OPT2 which will employ a double scan of program loops identified by OPT1.

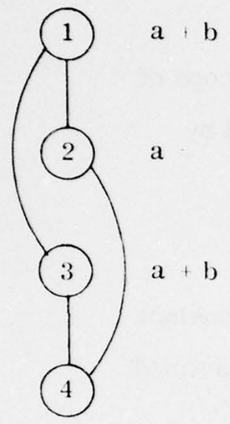
A.5 PARALLEL PATH OPTIMIZATIONS

The IF...THEN...ELSE and CASE constructions in HOLs produce parallel paths in the resulting program flow graph. Certain optimization possibilities arise because of this form of flow which are not addressed in the present JOCIT optimizer. The optimizations considered in the following sections are:

- Improved forward flow analysis
- Load promotion/store delay
- Common name recognition

A.6 IMPROVED FORWARD FLOW ANALYSIS

In the current JOCIT optimizer, all forward branches are treated as conditional insofar as their effect on the state of the search set is concerned. For example, in the following program segment:



the assignment to "a" in Block 2 prevents "a+b" in Block 3 from being recognized as common with that in Block 1 although they compute the same value. The analysis can be improved in OPT2 by restoring search set value numbers at block entrances whose immediate textual predecessor block exits by an unconditional branch.

A.6.1 LOAD PROMOTION/STORE DELAY

In any multipath graph, code space can be saved by preloading common values and by delaying common stores as the following simple case demonstrates:

```
IFEITH BOOL$
    AA($I+J$) = KK*5$
        ORIF 1$
            AA($I+J$) = KK*5+1$
```

In the current JOCIT compiler, if neither I+J nor KK*5 appear before the IFEITH, the code (shown for the HIS-6000) will be:

```
            LDA     BOOL
            TZE     L1
            LDQ     I
            ADQ     J
            EAX0    0,QL
            LDQ     KK
            MPY     5,DL
            STQ     AA,X0
            TRA     L2
    L1      LDQ     I
            ADQ     J
            EAX0    0,QL
            LDQ     KK
            MPY     5,DL
            ADQ     1,DL
            STQ     AA,X0
    L2      ...
    ------------------------
    Total   16
```

Applying the diamond optimizations, the code would be:

```
            LDQ     I
            ADQ     J        PRECOMPUTE I+J
            EAX0    0,QL     SAVE IN X0
            LDQ     KK
            MPY     5,DL     PRECOMPUTE KK*5--
                             SAVE IN Q
            LDA     BOOL
            TNZ     L2       TRUE CASE NOW NULL!
    L1      ADQ     1,DL     FALSE CASE:
                             COMPUTE KK*5+1
    L2      STQ     AA,X0    DELAYED STORE OF
                             AA($I+J$)
    -------------------------------------------
    Total   9                Reduction = 44%
```

A.7 NAME COMMONALITY

The current JOCIT optimizer employs the value-folding technique to implement common expression recognition. This excludes recognition of certain cases in which expressions are computed on parallel paths which are formally common but which may compute different values. The following program example demonstrates:

```
IFEITH BOOL$
    PP(XX+YY)$
        ORIF 1$ BEGIN
            XX=XX+1$ PP(XX+YY)$
                END
                    ...XX+YY...
```

In this example, the expression XX+YY after the IFEITH/ORIF construction is formally common with the XX+YY computed on each path of the diamond; however, since it computes different values--XX is redefined on the ORIF path--the optimizer will not recognize the common case. This problem may be solved within the present value-folding design by the following technique:

- At the terminus of parallel paths, all live names and expressions are permuted for formal name matching. This involyes examining the value synonym list and substituting the different name synonyms until the list is exhausted or a match occurs on both paths. In practice, synonym lists for values are quite short, so that this process is not prohibitively slow.
- Common names and expressions so recognized are then assigned to temps on the parallel paths. (e.g., XX+YY is assigned to T1 on both paths in the above example). The same temp is used as a surrogate name for the common expression. This is to give the

expression a common residence on both paths which the code generator may subsequently assign to a register in order to achieve the desired optimal effect.

- Common names and expressions thus recognized are now posted to the search set in the usual manner, and the temp is also posted as a synon 'm for the posted value. Thus, subsequent occurrences of the expression are found common through the conventional folding technique. Dead variable analysis may delete the assignments to the temps on the parallel paths. The optimization is realized by the code generator when it is called to compute the value common subsequent to the parallel network; the temp (which may have been register-dedicated on each path) is chosen as the optimal source of the value.

In the example given above, the value XX+YY is stored in a temp, the same temp, on each path in order for it to be passed as a value parameter in the calls to PP. Thus, references to XX+YY after the diamond may be replaced by references to the temp.

A.8 USE OF COMPOOL BY THE OPTIMIZER

A.8.1 REDEFINED VARIABLE LISTS FOR COMPOOL DEFINED PROCEDURE

Optimization may be enhanced through extension of the COMPOOL concept. The greatest potential payoff is in refining the spoil analysis at COMPOOL-defined procedure calls. Another useful application is in the use of branch frequency information to improve certain loop optimizations.

In the present JOCIT optimizer, calls to local procedures invoke a spoiling process in which only those variables global to the called procedure and also to the point of call are assumed to be redefined by the call. This process is mechanized in OPT1 which constructs a list of global variables redefined by the procedure. The list also includes other procedures called, so that cascaded effects are accommodated. For calls to externally-defined or COMPOOL-defined procedures, the optimizer makes the assumption that all external and common data are spoiled by the call. This is clearly a safe but overly conservative assumption.

An improvement may be experienced by appending redefined variable list information to the COMPOOL entry for each procedure during COMPOOL assembly. This requires one or both of two other changes to the COMPOOL process. The first requires the programmer to specify in the COMPOOL declaration for the procedure those global variables assigned and those external procedures called. The second approach requires that the COMPOOL source for procedures include the entire procedure body. In this case, OPT1 would be called as part of the COMPOOL assembly, and its constructed RVL would be output with the procedure entry in the COMPOOL.

A.8.2 BRANCH FREQUENCY DATA

As part of the integrated approach to the LCF design, execution measurement data may be subjected to postmortem reduction and the branch frequency data entered into the program COMPOOL. The branch points would be identified by statement number. Thus, subsequent compilations of the program would provide for access to the branch frequency data in that program's COMPOOL by the optimizer for the purposes of replacing branch frequency assumptions by actual operational experience. A limitation inherent in this approach is that source program modifications to the measured program are prohibited between any measured execution and a subsequent recompilation. This guarantees that the statement number data is consistent between the COMPOOL and the subsequent compilation.

A.9 MISCELLANEOUS OPTIMIZATION ENHANCEMENTS

A.9.1 VALUE USAGE CONTEXT

The optimizer will set bits in each VALU entry in the IL to indicate the contexts in which the value is used. These bits may distinguish between computational usage and subscript usage, for example, and will aid the code generator in register allocation for values. For example, a value having only subscript uses may be profitably allocated to an index register. The current JOCIT code generator already makes use of this information, so the setting of these bits yields a low-cost object code efficiency improvement.

A.9.2 UNREFERENCED PROCEDURE DELETION

It frequently happens that as a program ages, certain procedures are no longer referenced, and the programmer fails to remove them. This is an easy case for the compiler to recognize, and the unreferenced procedure need not be compiled.

A.9.3 SINGLE-REFERENCE PROCEDURE OPTIMIZATION

A procedure with only one reference may perhaps be more efficiently compiled as an open routine with actual parameters substituted for formal parameters. This substitution may be performed by OPT2 (or perhaps the straightener) by collecting the actual parameter IL code and substituting it for the corresponding formal parameter IL code, and by eliminating the procedure prologue and epilogue.

A.9.4 REFINED REGION DEFINITION

That segment of the program optimized by the JOCIT compiler in a single unit is known as a region. Currently it is the case that a region break occurs when the optimizer exhausts working storage for the region. When this happens, the

IL is flushed, and working space is recovered for the next region. In the production compiler, working space is sufficient to permit the optimization of several hundred source lines per region.

CSC proposes that the region end definition be refined to prevent the termination of a region within a loop that might otherwise be contained. This can be accomplished by the straightener, which will record for each loop entry in the loop list the number of IL entries associated with the loop body. OPT2 may then estimate the working storage requirements - based on the IL count for the loop - whenever a loop top is seen. and terminate the region before the loop in the event that working space is insufficient. This will permit the optimization of a loop, or nest of loops, within a single region, which the termination of a region in mid-loop prevents (e.g., after such a region end, no strength reduction or redistribution may occur).

A.9.5 REASSOCIATION, DISTRIBUTION, AND CONSTANT COLLECTION

Subscript expressions, linearized by the analysis phase, may be more completely optimized if the rules of reassociation, multiply distribution, and constant reordering are applied. These rules, to be implemented in OPT2, are summarized as follows ("K" stands for any constant, and "&" stands for any associative operator*):

<u>Reassociation</u>

1. The expression $(A \& K_1) \& K_2$ is changed to $A \& K_3$, where $K_3 = K_1 \& K_2$
2. The expression $(A \& K) \& B$ is changed to $(A \& B) \& K$
3. The expression $(A \& K_1) \& (B \& K_2)$ is changed to $(A \& B) \& K_3$, where $K_3 = K_1 \& K_2$

*Add and multiply are the only associative operators of interest in this discussion.

Distribution of Multiply

4. The expression $(A + K_1) * K_2$ is changed to $(A * K_2) + K_3$, where $K_3 = K_2 * K_1$

Constant Reordering

5. The expression (K & A) is canonically reordered to (A & K)

The effect of these can be seen on the following two-dimensional array reference XX($1+I, J+1$), where XX is a 10-by-10 array. The linearized subscript is (1+I) + (J+2)*10. The successive application of the above rules to the triads as seen left-to-right is shown:

(1+I) becomes (I+1) by Rule 5.

(J+2)*10 becomes (J*10)+20 by Rule 4.

(I+1)+((J*10)+20) becomes (I+(J*10))+21 by Rule 3.

One of the primary effects of these rules is that any constant offset is floated to the right where it is removed by the code generator and placed in the address field as an address offset.

APPENDIX B - GLOSSARY

ALGOL	Algorithmic Language. There are two versions, ALGOL 60 and ALGOL 68. ALGOL is used as a publication standard and by universitites. First use of GENESIS system was for the ALGOL 60 language employing the same compiler on five different hosts generating code for six different targets.
ANZR	Machine and Language-Independent Syntax Analyzer Program. ANZR is the canonical analyzer program that processes source language statements according to the syntax tables generated by the GENESIS system for a particular HOL. ANZR is employed in both syntax passes of the JOCIT compiler and in the syntax analysis phase of the COMPOOL assembler.
CDD	Composite Debugging Dictionary. The CDD is proposed in this report as an output of the HOL compiler to be used by the recommended statistics collection, program validation, and program debugging tools.
CF	Code File. This file is used as the interface in the JOCIT compiler between the preset processor and editor phases and also between the code generator and editor phases. It contains an encoded representation of both object code data and instructions. The editor phase converts the code file to actual object module format for the target machine.
Construct String	The file of language tokens operated on by the ANZR program within the analysis phases of the JOCIT compiler. First generated by the Precognition program, the construct string is analyzed and modified by Pass 1 analysis to recognize data

	declarations. Executable statement tokens are then passed in the construct string as an intermediate file to Pass 2 analysis for generation of the IL file. The Pass 2 process applies the rules of the GENESIS syntax tables to recognize statements and in the process reduces the string until it is exhausted.
Cradle	The JOCIT compiler executive component that provides all host system/compiler interfaces, such as I/O services, phase loading, etc. It is a collection of functionally independent modules that must be changed when rehosting the JOCIT program.
CSTS	Computer Sciences Teleprocessing System. This is the time-sharing system operating on the UNIVAC 1108 as the program service for the INFONET Division of Computer Sciences Corporation.
DD-path	Decision-to-Decision Path. A term used in the JAVS documentation to describe the domain of a program between one conditional branch and a subsequent one. Program validation is described, in part, in terms of the exercise of all of the DD-paths in the program.
GCOS	This is the operating system for the HIS-6000 series of machines. It provides both batch and interactive/timesharing services. This is the system under which JOCIT and JAVS are presently implemented.
GENESIS	A system developed by Computer Sciences Corporation for describing the syntax of HOLs. The output of GENESIS is a set of syntax tables which encode the described syntax. These

tables are then incorporated permanently into a compiler and are processed by the standard ANZR program in parsing the source language input to the compiler. The GENESIS approach is employed in the JOCIT compiler. It currently resides on the INFONET 1108/CSTS, and future plans call for its rehosting to GCOS.

GMAP	The standard assembly language and macro processor for the HIS-6000 series.
HOL	High-Order Language. An abbreviation for any number of procedure-oriented programming languages above the assembly language level.
IL	Intermediate Language. In this report, IL refers to the file generated by the analysis phase of the JOCIT compiler which contains an encoded representation of the executable program structure of a compiled module. It is generated in RPN form and is the input to the code generator which translates the IL into machine code. When optimization is selected, the IL is processed and transformed by the optimizer before it reaches the code generator. Thus, all global optimization and source local optimization are expressed in the IL file.
INFONET	The Information Network, a division of Computer Sciences Corporation, which operates the 1108/CSTS service.
JAVS	JOVIAL Automated Verification System. JAVS was developed by RADC to assist in comprehensive and systematic testing of JOVIAL/J-3 programs.

JCVS	JOVIAL Compiler Validation System. JCVS was developed by RADC to provide a standard for measuring all JOVIAL compilers.
JLMT	JOVIAL Language Measurement Tool. A term used to describe a statistics-gathering methodology for the JOVIAL/73 language.
JOCIT	JOVIAL Compiler Implementation Tool. Developed by RADC to provide a standard compiler for the JOVIAL/J-3 Language which employs extensive optimization and which permits convenient rehosting and retargeting of J-3 compilers.
JOVIAL	Jules' Own Version of the International Algebraic Language. Developed at SDC during the late 1950s and early 1960s, JOVIAL has been widely used as a standard for military command/control systems. Several dialects exist. The military standard is J-3, described in Air Force Manual AFM 100-24. The JOCIT compiler implements the full J-3 language. A major evolutionary development of JOVIAL is the J73 language whose final definition appeared in 1975. The only known implementation of the Level I subset of J73 is on the DEC-10 system developed for Wright-Patterson AFB, AFAL (Air Force Avionics Laboratory).
JXEC	The component name of the principal compiler executive (also known as cradle) module of the JOCIT compiler.
LCF	Language Control Facility.
MONITOR	JOVIAL/J-3 statement that implements run-time tracing of assignments to the variable named in the MONITOR statement.

MULTICS	Timesharing system developed at M.I.T. for the GE 645 system. A unique feature of MULTICS is that it is implemented almost entirely in a subset of PL/1.
Pragmatic Functions	A set of routines that support the analysis phases of the JOCIT compiler. The functions are invoked by the ANZR program when syntactic recognitions are made, and the rule governing the recognition names a pragmatic function. The function name as well as the recognition rule are encoded in the GENESIS-produced syntax tables which govern the analysis process.
Precognition	A part of the first analysis pass of the JOCIT compiler that performs primitive construct or token (reserved words, constants, comments, special marks, etc.) recognition.
RADC	Rome Air Development Center. RADC is the sponsor of this report and all of the existing tools described.
Regional Optimizer	Refers to the optimization scheme employed in the DEC-10 J73/I Compiler. It is a Level II LNRA (Linear Nested Region Analyzer) scheme in which the program is optimized in regions that are terminated by work space exhaustion or the occurrence of loop-top labels.
RPN	Reverse Polish Notation. This is the form of the IL produced by the analysis phase of the JOCIT compiler. It consists of a string of operands and operators in which the operands always precede the operator. Sometimes referred to as postfix notation. As opposed to infix notation (for example, (A+B)*C) where the operator is placed between the operands and where

	parentheses are required to express nonstandard procedure, RPN is parenthesis-free (for example, A B+C* is the RPN for the previously given infix notation).
SEMANOL	Semantics-Oriented Language. SEMANOL is a program developed under RADC sponsorship that permits the full syntactic and semantic specification of any programming language and supports both the analysis and interpreted execution of programs written in the language.
SOS	A line-oriented text editor operating under the DEC-10 timesharing system.
SYMPL	Systems Programming Language. SYMPL was developed by Computer Sciences Corporation as a compiler implementation language and used for the development of JOCIT.
Syntax Tables	The output of the GENESIS program. The syntax tables contain an encoded representation of the recognition rules presented as input to GENESIS.
TECO	Text Editing Program resident in both the DEC-10 and MULTICS systems.
TRACE	A JAVS directive that permits the run-time display of program variables as they are changed in value by assignment.
VALU	A term used to describe the value associated with any primitive or expression operand. The term is unique to the optimizer technology employed in the JOCIT Compiler.
WWMCCS	Worldwide Military Command and Control System.

APPENDIX C - BIBLIOGRAPHY

1. Anderson, E.R., Belz, F.C., Blum, E.K., "SEMANOL 73, A Metalanguage for Programming the Semantics of Programming Languages," TPR-TR-7, TRW Systems Group, Redondo Beach, California, 1974.

2. Lucas, P., et al., "Format Definition of PL/I," International Business Machines PL/I Definition Group of the Vienna Laboratory, Vienna, 1966.

3. Van Wijngaarden, A., et al., "Report on the Algorithmic Language ALGOL 68," Numerische Mathematik, 14, 79-218, Springer-Verlag, Berlin, 1969.

4. "JOCIT, JOVIAL Compiler Implementation Tool," *Final Report*, RADC-TR-74-322, Computer Sciences Corporation/Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1975. AD#A005307.

5. (Untitled Document) JOVIAL Automated Verification System (JAVS) Reference Manual.

6. (Untitled Document) JOVIAL Automated Verification System (JAVS) User's Guide.

7. "*JOVIAL J73 Compiler* Validation System, Class 1 Test Program User's Manual," Abacus Programming Corporation/Rome Air Development Center, Griffiss Air Force Base, New York, 1974.

8. "Methodology For Comprehensive Software Testing," Interim Report, RADC-TR-75-161, General Research Corporation/Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1975. AD#A013111.

9. "The Multiplexed Information and Computing Service: Programmer's Manual. Part I Introduction to MULTICS," Massachusetts Institute of Technology and Honeywell Information Systems Inc., 1972.

10. "The Multiplexed Information and Computing Service: Programmer's Manual. Part II Reference Guide to MULTICS," Massachusetts Institute of Technology and Honeywell Information Systems Inc., 1972.

11. "NCR Student COBOL Student Text," NCR Corporation, Dayton, 1975.

12. "A SEMANOL (73) Implementation Standard for JOVIAL (J73)," Final Technical Report, F30602-74-C-0067, TRW Systems Group/Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1975, RADC-TR-75-211, (A019768).

13. "SEMANOL (73) Reference Manual," N00123-74-C-1878, TRW Systems Group/Fleet Combat Direction Systems Support Activity, San Diego, 1975.

14. "SEMANOL (73) Specification of JOVIAL (J73)," Final Technical Report, RADC-TR-75-211, Vol. III (of four), TRW Systems Group/Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1975. AD#A019770.

15. "Standard Computer Programming Language for Air Force Command and Control Systems," Air Force Manual AFM 100-24, Department of the Air Force, Washington, D.C., 1967.

16. "A Standard for Language Implementation," Final Technical Report, RADC-TR-73-143, TRW Systems Group/Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1973. AD#765608/5.

17. "A Standard For Language Implementation, SEMANOL Reference Manual," TRW Systems Group, Redondo Beach, California, 1973.

18. "A Standard For Language Implementation, SEMANOL Specification of JOVIAL," TRW Systems Group, Redondo Beach, California, 1973.

19. "A Statistics Gathering Package For the JOVIAL Language," Final Technical Report, RADC-TR-73-381, Proprietary Software Systems, Inc./Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, New York, 1974. AD#775380/9GI.

☆U.S. GOVERNMENT PRINTING OFFICE: 1977-714-025/82

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible

D

E